Automated Chip Synthesis
User Guide

Version Y-2006.06, June 2006

Comments?
Send comments on the documentation by going to http://solvnet.synopsys.com, then clicking "Enter a Call to the Support Center."

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Arcadia, C Level Design, C2HDL, C2V, C2VHDL, Cadabra, Calaveras Algorithm, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, Hypermodel, iN-Phase, in-Sync, Leda, MAST, Meta, Meta-Software, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PowerMill, PrimeTime, RailMill, RapidScript, Saber, SiVL, SNUG, SolvNet, Superlog, System Compiler, TetraMAX, TimeMill, TMA, VCS, Vera, and Virtual Stepper are registered trademarks of Synopsys, Inc.

Trademarks (™)

Active Parasitics, AFGen, Apollo, Apollo II, Apollo-DPII, Apollo-GA, ApolloGAII, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BCView, Behavioral Compiler, BOA, BRT, Cedar, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, Davinci, DC Expert, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, DFM-Workbench, Direct RTL, Direct Silicon Access, Discovery, DW8051, DWPCI, Dynamic Model Switcher, Dynamic-Macromodeling, ECL Compiler, ECO Compiler, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Floorplan Manager, Formal Model Checker, FoundryModel, FPGA Compiler II, FPGA *Express*, Frame Compiler, Galaxy, Gatran, HANEX, HDL Advisor, HDL Compiler, Hercules, Hercules-Explorer, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSIM^{plus}, HSPICE-Link, i-Virtual Stepper, iN-Tandem, Integrator, Interactive Waveform Viewer, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Libra-Visa, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Metacircuit, Metamanager, Metamixsim, Milkyway, ModelSource, Module Compiler, MS-3200, MS-3400, Nova Product Family, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Nova-VHDLlint, Optimum Silicon, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Polaris-CBS, Polaris-MT, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Protocol Compiler, PSMGen, Raphael, Raphael-NES, RoadRunner, RTL Analyzer, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, SmartModel Library, Softwire, Source-Level Design, Star, Star-DC, Star-MS, Star-MTB, Star-Power, Star-Rail, Star-RC, Star-RCXT, Star-Sim, Star-SimXT, Star-Time, Star-XP, SWIFT, Taurus, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Venus, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (^{SM})

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
All other product or company names may be trademarks of their respective owners.

Automated Chip Synthesis User Guide, version Y-2006.06

Contents

3. Customizing Automated Chip Synthesis

Preface

This preface includes the following sections:

What's New in This Release

This section describes the changes included in Automated Chip Synthesis version Y-2006.06.

Changes

The following Automated Chip Synthesis variables are obsolete as of the Y-2006.06 release:

- Batch submission command variables

  The following variables are obsolete:

  - `acs_use_lsf`
  - `acs_bsub_args`
  - `acs_bsub_exec`

  To control batch submission, use the `acs_submit` and `acs_submit_large` commands, as described in "Specifying the Batch Submission Command" on page 3-45.

- Physical Compiler variables

  Automated Chip Synthesis no longer supports Physical Compiler. The following variables are obsolete:

  - `acs_congestion_report_suffix`
  - `acs_pdef_suffix`
  - `acs_psyn_exec`
  - `acs_spef_suffix`

- DC-FPGA variables

  Automated Chip Synthesis no longer supports DC-FPGA. The following variable is obsolete:

  - `acs_fpga_exec`

- DB dcsh mode variables

  Automated Chip Synthesis no longer supports Design Compiler DB dcsh mode. The following variable is obsolete:

  - `acs_script_mode`

- Design budgeting variables

  Automated Chip Synthesis no longer supports PrimeTime design budgeting; all budgeting is performed by Design Compiler budgeting. The following variables are obsolete:

  - `acs_bs_exec`
  - `acs_budget_output_file_suffix`
  - `acs_budget_script_file_suffix`
  - `acs_budgeting_var`
  - `acs_use_dc_gate_level_budgeting`

Known Limitations and Resolved STARs

Information about known problems and limitations, as well as about resolved Synopsys Technical Action Requests (STARs), is available in the *Design Compiler Release Notes* in SolvNet.

To see the *Design Compiler Release Notes*,

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNet.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)
3. Click Release Notes in the Main Navigation section (on the left), click Design Compiler, then click the release you want in the list that appears at the bottom.

About This Guide

This guide describes the methodology and commands used to run the Automated Chip Synthesis feature.

Unless otherwise specified, all features discussed in this manual are available in both XG mode and DB mode. Features that are available only in a particular mode are marked as such.

Additionally, all examples presented in this manual work in both XG mode and DB mode. When the command syntax is the same in both XG mode and DB mode (dctcl command language), the manual provides a single example, preceded with the dc_shell-xg-t> prompt. When the command syntax differs, the manual provides different examples, as appropriate.

Audience

This user guide is for engineers who use Design Compiler for compiling designs. You need to be familiar with

- Logic design principles
- Design Compiler tools
- The UNIX operating system
- The tool command language (Tcl)

Related Publications

For additional information about Automated Chip Synthesis, see

- Synopsys Online Documentation (SOLD), which is included with the software for CD users or is available to download through the Synopsys Electronic Software Transfer (EST) system
- Documentation on the Web, which is available through SolvNet at http://solvnet.synopsys.com
- The Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at http://mediadocs.synopsys.com

You might also want to refer to the documentation for the following related Synopsys products and features:

- Design Compiler
- (V)HDL Compiler
- Design budgeting

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
`Courier bold`	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[ ]	Denotes optional parameters, such as *`pin1`* [*`pin2 ... pinN`*]
|	Indicates a choice among alternatives, such as `low | medium | high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call With the Support Center.”

To access SolvNet,

1. Go to the SolvNet Web page at http://solvnet.synopsys.com.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)

If you need help using SolvNet, click HELP in the top-right menu bar or in the footer.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a call to your local support center from the Web by going to http://solvnet.synopsys.com (Synopsys user name and password required), then clicking “Enter a Call With the Support Center.”
- Send an e-mail message to support_center@synopsys.com.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

1

Introduction to Automated Chip Synthesis

As designs become larger and larger, the compile runtimes might become intolerable. When the top-down compile runtime of a large, hierarchical design exceeds your requirements, you can reduce runtime by compiling portions of your design in parallel. This technique is referred to as a divide-and-conquer strategy. Although this technique reduces runtime, it increases the data management requirements.

The Automated Chip Synthesis tool addresses these challenges by providing a higher level of abstraction for synthesis, including compile strategy implementation and automatic generation of compile scripts and makefiles.

Automated Chip Synthesis is a set of commands that provide a data management environment, automate the divide-and-conquer synthesis flows, and enable parallel compilation of the hierarchical design. You can use Automated Chip Synthesis with Design Compiler.

This chapter includes the following sections:

Product Overview

To use Automated Chip Synthesis, you must

- Have the DC Expert or DC Ultra product (depending on the flow you are using)
- Provide top-level constraints for the design (or subdesign) being synthesized

Automated Chip Synthesis implements the divide-and-conquer synthesis strategy by using the concept of compile partitions. Compile partitions are the subdesigns on which Automated Chip Synthesis performs parallel top-down compiles. Each compile partition should be as large as possible (to maximize the quality of results), without exceeding your runtime tolerance.

Automated Chip Synthesis provides a chip-level HDL input command (`acs_read_hdl`) and chip-level compile commands (`acs_compile_design`, `acs_refine_design`, and `acs_recompile_design`). These commands perform all of the steps necessary to compile your design using the divide-and-conquer synthesis strategy.

Automated Chip Synthesis provides the following capabilities:

- Reads the HDL source code for the entire design

  Automated Chip Synthesis analyzes the source code, then elaborates and links the top-level design.

- Selects the compile partitions

  Automated Chip Synthesis can select the compile partitions, or you can explicitly define them.

- Generates logical constraints for the compile partitions

  As with a top-down compile of the design, you provide the top-level constraints for the design. Automated Chip Synthesis uses these top-level constraints to generate constraints for the compile partitions.

  Automated Chip Synthesis supports multiple technologies to generate the logical partition constraints. The technology used depends on the state of your design.

  - RTL budgeting or top-down environment propagation generates the partition constraints for unmapped designs.
  - Design budgeting generates the partition constraints for mapped designs.

  If you do not want to use the constraints generated by Automated Chip Synthesis, you can provide an override constraint file.

- Generates the compile scripts

  Automated Chip Synthesis automatically generates dctcl-format compile scripts for each compile partition and the top-level design. You can use a default compile strategy, or you can customize the compile strategy (by using design attributes or a custom script file) based on your requirements.

  The generated compile scripts provide error handling to stop the compile process when serious errors occur. You can customize the error handling to meet your requirements.

  You can choose between the standard logic synthesis flow (`compile` command) and the ultra logic synthesis flow (`compile_ultra` command).

If you do not want to use the compile script generated by Automated Chip Synthesis, you can provide a custom compile script.

- Generates a makefile to manage the parallel compile process

  The makefile charts the dependencies between the compile partitions to ensure that a valid chip-level compile occurs.

  Automated Chip Synthesis supports parallel compiles by providing interfaces to UNIX gmake (or other parallel make utilities) and to the Load Sharing Facility (LSF) software from Platform Computing. Automated Chip Synthesis runs all compile jobs on the same platform. The platform used to initiate the compile jobs determines the platform used to run the compile jobs.

  Note:

  For information about the LSF software, see the Platform Computing Web site at http://www.platform.com.

  You can run the makefile within the command shell or from the UNIX prompt.

- Supports incremental updates to the design

  Automated Chip Synthesis enables you to make small changes to parts of your design without requiring you to recompile the complete design.

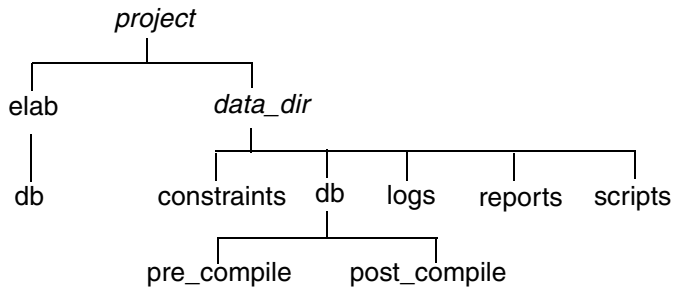
- Manages the design data

  Automated Chip Synthesis provides a customizable directory structure for managing the design data. The Automated Chip Synthesis commands automatically create and maintain data directories for each chip-level compile that you perform.

Data Management Overview

By default, Automated Chip Synthesis uses the directory structure shown in Figure 1-1 to manage the design compilation data. The top-level directory (named *project* in Figure 1-1) is called the project root directory. When you run Automated Chip Synthesis on a design, you must invoke the synthesis tool from the project root directory.

Figure 1-1 Default Directory Structure



For information about customizing the directory structure, see "Customizing the Directory Structure" on page 3-2.

Note:

If you want to customize the directory structure, you must do so (either in the .synopsys_dc.setup file or interactively) before you run the first chip-level compile command in a shell session. Automated Chip Synthesis ignores any changes you make to the directory structure after you have run a chip-level compile command.

Methodology Overview

To use Automated Chip Synthesis,

1. Invoke Design Compiler.

Invoke the synthesis tool from the project root directory. Automated Chip Synthesis performs several setup tasks when you invoke the synthesis tool. For details about this step, see “Invoking the Synthesis Tool” on page 2-2.

2. Prepare the design.

   Use the `acs_read_hdl` command to read the HDL source code for the entire design. The result of this command is a generic technology (GTECH) design database. To complete design preparation, apply the top-level constraints to this GTECH design database. For details about this step, see “Preparing Your Design” on page 2-3.

3. Compile the design.

   Use the `acs_compile_design` command to compile the GTECH design database. For details about this step, see “Compiling Your Design” on page 2-15.

4. Analyze the results.

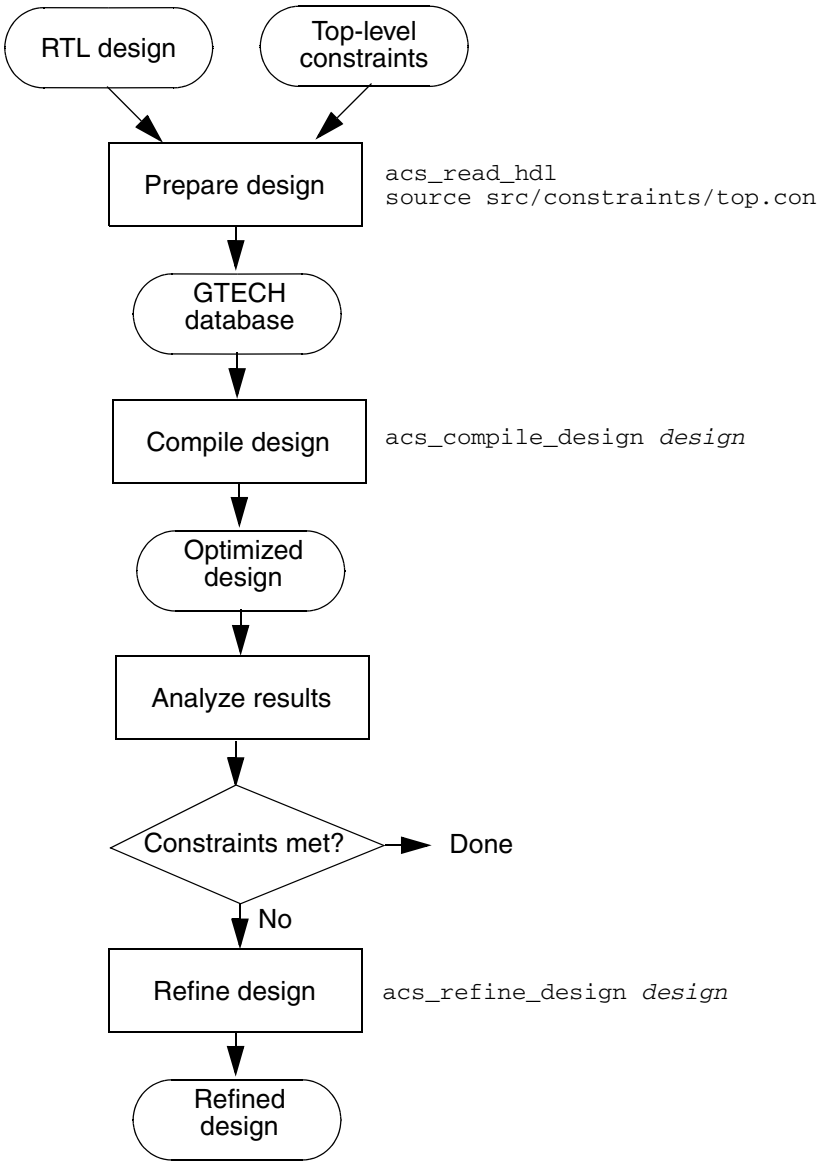
   Automated Chip Synthesis generates reports that you can use to analyze the resulting design. For details about this step, see “Analyzing the Results” on page 2-31.

5. Refine the design, if necessary.

   If your design does not meet constraints, use the `acs_refine_design` command to refine the design. If necessary, you can repeat this step until your design meets the constraints. For details about this step, see “Refining Your Design” on page 2-35.

Figure 1-2 shows a flowchart of this process.

Figure 1-2 Automated Chip Synthesis Design Flow



Known Limitations

Automated Chip Synthesis has the following limitations:

- Automated Chip Synthesis does not support the `update_lib` command.

  Your custom scripts cannot include this command. To work around this limitation, run the `update_lib` command in dc_shell, then save the library by using the `write_lib` command.

- Top-down environment propagation does not support old Synopsys database format (.db) files.

  If your .db file was generated using version 1997.01 or an earlier version, regenerate the .db file before running Automated Chip Synthesis.

Limitations discovered after the publication of this document are listed in the *Design Compiler Release Notes* in SolvNet. For information about accessing the *Design Compiler Release Notes*, see “Known Limitations and Resolved STARs” on page xii.

2

Running Automated Chip Synthesis

This chapter defines how to run Automated Chip Synthesis. It assumes that you are using the default settings for all options. After you are familiar with the default flow described in this chapter, you can customize the flow (if necessary). For information about customizing Automated Chip Synthesis, see Chapter 3, "Customizing Automated Chip Synthesis."

This chapter contains the following sections:

Invoking the Synthesis Tool

You can run Automated Chip Synthesis from within DC Expert or DC Ultra (Tcl-based only). You invoke Design Compiler in XG mode by using the `dc_shell-xg-t` or `dc_shell-t` command.

Note:

To invoke Design Compiler in DB Tcl mode, use the `dc_shell-t -db_mode` command.

For more information about invoking Design Compiler, see the *Design Compiler User Guide*.

When running Automated Chip Synthesis, you must invoke the synthesis tool from the project root directory. When you invoke the synthesis tool from the project root directory, the tool automatically performs the following tasks:

- Executes the project setup file (.synopsys_dc.setup)

  At a minimum, the project setup file should define the technology library by setting the `link_library` and `target_library` variables.

- Sets the `acs_work_dir` variable to the project root directory, if this variable was not defined in the project setup file (Automated Chip Synthesis uses this variable to identify the project root directory)

Caution!

You should not set the `acs_work_dir` variable interactively. It must be set in the .synopsys_dc.setup file or by invoking the synthesis tool from the project root directory. Changing the `acs_work_dir` variable from the command line can cause problems with the Automated Chip Synthesis flow.

If you change the value of the `acs_work_dir` variable to a value other than the current directory, you must be careful when using relative paths in your setup variables (for example, the `search_path`, `target_library`, or `link_library` variables). Automated Chip Synthesis copies the setup variables to the env file used by the partition compile runs. When running the env file, the paths are referenced relative to the path specified in the `acs_work_dir` variable. You must ensure that the relative paths exist at the location specified by `acs_work_dir`.

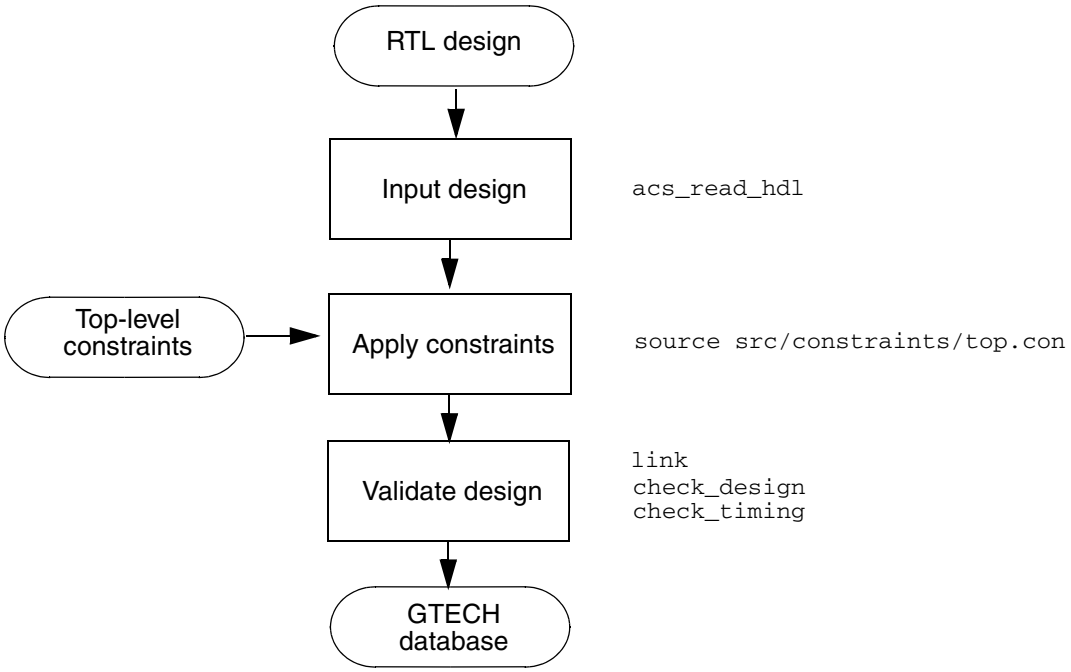
- Selects the shell for design compilation

  Automated Chip Synthesis compiles the design using the same shell that the chip-level commands were run under. For example, when you run Automated Chip Synthesis within Design Compiler in XG mode, the compile shell is dc_shell-xg-t.

Preparing Your Design

Figure 2-1 shows the process used to prepare a design for Automated Chip Synthesis.

Figure 2-1 Design Preparation



The following sections describe these tasks.

Inputting the RTL Design

Use the `acs_read_hdl` command to input the RTL design.

Note:

If your design has non-RTL source files, you must input them manually before running `acs_read_hdl`.

There are two modes for running the `acs_read_hdl` command:

- Update mode

  Update mode enables use of the incremental design update feature. In update mode, the `acs_read_hdl` command automatically saves the GTECH design database and other files

required for an incremental design update. Because it saves these files, update mode requires more disk space than standard mode.

- Standard mode

  Standard mode maintains backward compatibility with versions earlier than V-2004.06. In this mode, the `acs_read_hdl` command does not save the GTECH design database.

The following sections provide information about these modes.

Running acs_read_hdl in Update Mode

To run `acs_read_hdl` in update mode, specify the `-auto_update` option.

```
dc_shell-xg-t> file mkdir work
dc_shell-xg-t> define_design_lib DEFAULT -path work
dc_shell-xg-t> acs_read_hdl -auto_update top_design \
   -format fmt -hdl_source source_files
```

During the initial `acs_read_hdl` run, update mode behaves similarly to standard mode but saves the GTECH design database and other files required to perform an incremental design update. For information about incremental design updates, see “Updating Your Design” on page 3-58.

In update mode, the initial `acs_read_hdl` run performs the following tasks:

1. Identifies the updated HDL source files

   For information about how to specify the source files, see “Specifying the HDL Source Files” on page 2-8.

2. Reads the HDL source files

   In update mode, the `acs_read_hdl` command uses the `read_file` command to input the HDL source files.

   For VHDL source files, you can take advantage of the `elaborate -update` capabilities by setting the `acs_read_hdl_use_read_file_for_vhdl` variable to false (the default is true). When you set this variable to false, `acs_read_hdl` uses the `analyze` and `elaborate -update` commands to input VHDL files, rather than the `read_file` command. This variable affects the input of VHDL files only; in update mode, Verilog source files are always input using the `read_file` command.

   Important:

   The `-no_elaborate` option is not supported in update mode. If you specify `-no_elaborate` in addition to `-auto_update`, `acs_read_hdl` ignores the `-no_elaborate` option.

   For information about the input process, see "About the File Input Process" on page 2-10.

3. Links the top-level design

4. Saves the GTECH design database

   By default, the `acs_read_hdl -auto_update` command saves the GTECH design database in the $acs_work_dir/elab/db directory.

   You can change this location either by changing the directory naming conventions (see "Controlling Naming Conventions" on page 3-9) or by using the `-destination` option to specify the directory.

Running acs_read_hdl in Standard Mode

To run `acs_read_hdl` in standard mode, do not specify the `-update` or the `-auto_update` option.

```
dc_shell-xg-t> file mkdir work
dc_shell-xg-t> define_design_lib DEFAULT -path work
dc_shell-xg-t> acs_read_hdl top_design \
   -format fmt -hdl_source source_files
```

In standard mode, the `acs_read_hdl` command performs the following tasks:

1. Identifies the HDL source files

   For information about how to specify the source files, see "Specifying the HDL Source Files" on page 2-8.

2. Analyzes the HDL source files

   In standard mode, `acs_read_hdl` performs separate analysis and elaboration of the HDL source files, by using the `analyze` and `elaborate` commands.

   Important:

   If an error occurs during the analysis process, the `acs_read_hdl` command stops processing the source files.

   For information about the analysis process, see "About the File Input Process" on page 2-10.

3. Elaborates and links the specified top-level design (only for single-format RTL source files)

If your design has mixed-format RTL source files, the `acs_read_hdl` command does not elaborate or link the top-level design. In this case, you must manually elaborate and link the top-level design after running `acs_read_hdl`.

In addition, you can explicitly prevent elaboration (and linking), by using the `-no_elaborate` option.

This behavior differs from the elaboration behavior in update mode. For information about the update mode behavior, see "Running acs_read_hdl in Update Mode" on page 2-5.

In standard mode, the `acs_read_hdl` command does not save the GTECH design database; however, it is good practice to do so. By default, Automated Chip Synthesis uses the $acs_work_dir/elab/db directory to store the GTECH design database. To save the GTECH design database in this default location, enter

```
dc_shell-xg-t> write -format ddc -hierarchy \
   -output $acs_work_dir/elab/db/design.ddc
```

Note:

If you are running in DB mode, save the GTECH design database in .db format by using the `write -format db` command.

For information about changing the directory naming conventions, see "Controlling Naming Conventions" on page 3-9.

Specifying the HDL Source Files

You can specify the HDL source files in the following ways:

- Add the HDL source directories to the `search_path` variable.
- Specify the HDL source files or directories in the `acs_hdl_source` variable.

- Specify the HDL source files or directories in the `-hdl_source` option of the `acs_read_hdl` command.

Important:

The `-hdl_source` option overrides the `acs_hdl_source` variable, which overrides the `search_path` variable.

If you specify directory names, the `acs_read_hdl` command analyzes all files in the specified directory whose file extensions match the file extensions specified by the `acs_verilog_extensions` variable (default is .v) or the `acs_vhdl_extensions` variable (default is .vhd). You can restrict the files considered by the `acs_read_hdl` command by using the `-format` option. If you specify `-format verilog`, only those files matching the `acs_verilog_extensions` values are analyzed. If you specify `-format vhdl`, only those files matching the `acs_vhdl_extensions` are analyzed.

If you are using a hierarchical directory structure to store the source files, specify the `-recurse` option to traverse through the directory tree.

To simplify the RTL input process, store only source files, Verilog include files, and VHDL packages in the source directories, and do not provide multiple source files for the same module. Place VHDL configuration files in a separate directory. If you must place nonsource files in a source directory, use the `-exclude_list` option to specify these excluded files. You can also use the `acs_exclude_list` and `acs_exclude_extensions` variables to exclude files.

Locating Verilog Include Files. Like HDL Compiler, Automated Chip Synthesis uses the search path to locate Verilog include files and uses the first file it finds as the include file.

However, if it does not find the include file in the search path, Automated Chip Synthesis also searches the HDL source directories (as specified by the `-hdl_source` option or the `acs_hdl_source` variable). If Automated Chip Synthesis locates a single copy of the include file in the HDL source directories, Automated Chip Synthesis adds the directory to the search path, so that HDL Compiler can analyze the design.

If Automated Chip Synthesis does not find the include file, or if it finds multiple copies of the include file in the HDL source directories, it generates a warning message. In such cases, HDL Compiler cannot analyze the design.

Defining Verilog Macros. The `acs_read_hdl` command allows you to define Verilog macros during analysis, similar to the `analyze -define` command. You specify the macros to be defined by setting the `acs_hdl_verilog_define_list` variable before running the `acs_read_hdl` command.

About the File Input Process

By default, the `acs_read_hdl` command establishes the order in which to input the HDL source files by taking into account the Verilog include files and the VHDL packages, in addition to the order in which you specified the source files. To force the `acs_read_hdl` command to input the HDL source files in the order you specified (from left to right), use the `-no_dependency_check` option.

The `acs_read_hdl` command ignores source code delimited by the `translate_off` and `translate_on` compiler directives (in both Verilog and VHDL source files) or by the `synthesis_off` and `synthesis_on` compiler directives (VHDL only). In Verilog source files, the compiler directives are comments that start with the `synopsys` keyword. In VHDL, the compiler directives are comments

that start with the `synopsys` or `pragma` keywords. When processing Verilog source files, you can disable compiler directive support in Verilog by setting the `hdlin_translate_off_on` variable to false. When processing VHDL source files, you can use the `hdlin_pragma_keyword` variable to specify an additional keyword to identify compiler directives.

You can use the `acs_read_hdl` command to analyze VHDL packages by specifying the `-library` option. When you use this option, the `acs_read_hdl` command analyzes the packages into the specified library and does not perform elaboration. You must have previously defined the library by using the `define_design_lib` command.

Applying the Top-Level Constraints

At a minimum, the top-level constraint file must contain the following constraints:

- Input delays
- Output delays
- Clocks
- Exceptions (false paths, multicycle paths)

Validating Your Design

Automated Chip Synthesis compiles your design in batch mode. For best results, perform the following validation tasks before running Automated Chip Synthesis:

- Run the `check_design` command to validate that the design database is consistent.
- Run the `check_timing` command to validate that the design is properly constrained.

For more information about the `check_design` and `check_timing` commands, see the Design Compiler documentation.

Generating Verification Setup Files

By default, Automated Chip Synthesis generates verification setup files. These files contain setup information for Formality that is based on the design transformations that occur during the design preparation and compile processes.

Automated Chip Synthesis generates the following verification setup files:

- A top-level verification setup file that captures the design transformations that occur during design preparation and budgeting

  By default, this file is saved in the $acs_work_dir/default.svf file. To change the name of this file, use the `set_svf` command.

  ```
  dc_shell-xg-t> set_svf top.svf
  ```

  Note:

  Automated Chip Synthesis generates the top-level verification setup file only if transformations occur during design preparation that must be communicated to Formality.

- A verification setup file for each compile partition that captures the design transformations that occur for that partition during the compile process

  The partition verification setup files are named *partition*.svf, where *partition* is the partition name. By default, Automated Chip Synthesis saves these files in the $acs_work_dir/*dest_dir*/svf directory.

Important:

Reusing a destination directory can result in invalid verification setup files. Always use a new destination directory for subsequent chip-level compile runs.

To read the generated verification setup files into Formality, enter the following command in Formality (assuming that the project root directory is your current working directory and that the default.svf file exists):

```
fm_shell> set_svf {./default.svf ./dest_dir/svf}
```

Because the top-level verification setup file is not always generated, you might want to use a conditional statement to set the verification setup file in Formality. For example,

```
fm_shell> if {[file exists ./default.svf]} {
   set_svf {./default.svf ./dest_dir/svf}
} else {
    set_svf {./dest_dir/svf}
}
```

To disable verification setup file generation, run the `set_svf -off` command. If you place this command at the top of your .synopsys_dc.setup file, no verification setup files are generated. Otherwise, Automated Chip Synthesis generates verification setup files for the design transformations that occur before you run the `set_svf -off` command.

Compiling Your Design

Use the `acs_compile_design` command to compile the GTECH design database.

```
dc_shell-xg-t> acs_compile_design design
```

Automated Chip Synthesis compiles the design using the same shell (dc_shell-xg-t or dc_shell-t) as the shell from which you ran the `acs_compile_design` command.

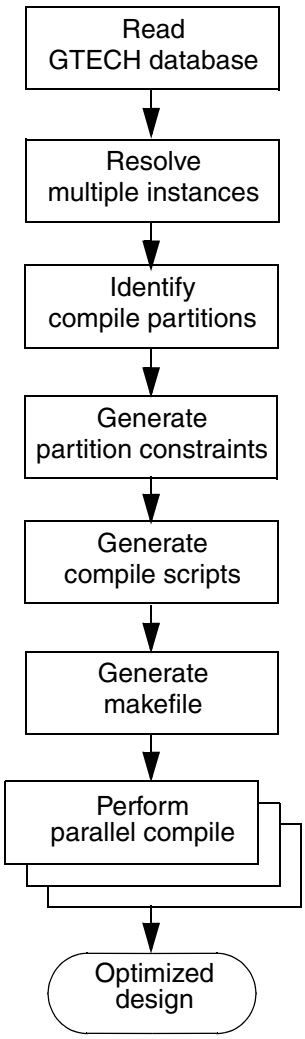
By default, Automated Chip Synthesis stores the data generated by the `acs_compile_design` command in the $acs_work_dir/pass0 directory. You can use another destination directory by specifying the `-destination` *`dest_dir`* option when you invoke the `acs_compile_design` command. For information about changing the directory naming conventions, see “Controlling Naming Conventions” on page 3-9.

The `acs_compile_design` command performs the tasks shown in Figure 2-2.

Note:

If a makefile exists in the destination directory ($acs_work_dir/*dest_dir*/Makefile), the `acs_compile_design` command performs only the parallel compile task (unless you specify the `-force` option).

Figure 2-2 Compile Design Flow



The following sections describe these tasks.

Reading the GTECH Database

The `acs_compile_design` command first checks for the specified design in memory. If it does not locate the design in memory, it reads the GTECH database file from its default location (elab/db). If you are running in XG mode, the `acs_compile_design` command looks for the *design*.ddc file in the $acs_work_dir/elab/db directory. If you are running in DB mode, the `acs_compile_design` command looks for the *design*.db file in the $acs_work_dir/elab/db directory.

If your GTECH database file does not follow this naming convention, use one of the following methods to read the GTECH database:

- Read the design into memory before running the `acs_compile_design` command.
- Change the default location for the GTECH database file.

  For information about changing default file locations, see "Customizing the Directory Structure" on page 3-2.

Resolving Multiple Instances

By default, Automated Chip Synthesis resolves multiple instances in your GTECH design by selecting a master instance to represent each set of instances having the same reference design.

Caution!

If an instance has constant logic on one or more input ports (or if one or more input ports are unconnected), you cannot use a master instance to resolve the multiple instances. In these situations, you must resolve the multiple instances during design

preparation by uniquifying the design. For information about uniquifying the design and other methods of resolving multiple instances, see “Resolving Multiple Instances” on page 3-13.

Automated Chip Synthesis selects the instance with the smallest (alphanumerically) name as the default master instance. For example, if you have a design that contains two instances U1 and U2 of design A, Automated Chip Synthesis selects U1 as the master instance.

When generating partition constraints for the GTECH design, Automated Chip Synthesis uses only the master instance to generate constraints for the reference design.

Identifying Compile Partitions

Automated Chip Synthesis uses the following designs as the compile partitions:

- First-level subdesigns (hierarchical children of the top-level design)
- Reference designs of multiple instances

Note:

If a design has a `dont_touch` attribute, Automated Chip Synthesis does not use that design as a compile partition.

If you want different (or additional) compile partitions, see “Partitioning the Design” on page 3-16 for information about setting compile partitions.

The `acs_compile_design` command writes the top-level design and the compile partitions to the $acs_work_dir/*dest_dir*/db/pre_compile directory.

Generating Partition Constraints for GTECH Designs

Automatic Chip Synthesis generates logical constraints for the compile partitions from the top-level design constraints.

Generating Logical Partition Constraints

Automated Chip Synthesis can use the following methods to generate logical constraints for the GTECH compile partitions:

- RTL budgeting

  RTL budgeting allocates timing and environment constraints among the compile partitions by taking into account the portion of the total delay required by each partition.

- Top-down environment propagation

  Top-down environment propagation propagates the design environment, constraints, and attribute settings of the top-level design to the compile partitions. It uses default values for input and output delays on the blocks. Top-down environment is faster and uses less memory than RTL budgeting; however the results are not as accurate.

By default, Automated Chip Synthesis uses RTL budgeting to generate the partition constraints for GTECH designs. To use top-down environment propagation instead, set the `acs_use_default_delays` variable to true.

When generating partition constraints, the `acs_compile_design` command performs the following tasks for each compile partition:

1. Executes the `reset_design` command on the subdesign
2. Generates constraints for the subdesign
3. Writes the generated constraints to *dest_dir*/constraints/*partition*.con

RTL Budgeting. Table 2-1 lists the types of constraints generated for each partition.

Table 2-1 RTL Budgeting Details

Type of information	Constraint commands
Operating conditions	`set_operating_conditions`

Wire load models	`set_wire_load_min_block_size`
	`set_wire_load_mode`
	`set_wire_load_model`
	`set_wire_load_selection_group`

System interface	`set_driving_cell`

Design rule constraints	`set_max_capacitance`
	`set_max_fanout`
	`set_max_transition`
	`set_min_capacitance`
	`set_min_fanout`

Table 2-1 RTL Budgeting Details (Continued)

Type of information	Constraint commands

Timing constraints	`create_clock`
	`group_path`
	`set_clock_gating_check`
	`set_clock_latency`
	`set_clock_transition`
	`set_clock_uncertainty`
	`set_critical_range`
	`set_false_path`
	`set_input_delay`
	`set_max_delay`
	`set_max_time_borrow`
	`set_min_delay`
	`set_multicycle_path`
	`set_output_delay`
	`set_propagated_clock`
	`set_resistance`
	`set_timing_ranges`

Logic assignments	`set_case_analysis`
	`set_equal`
	`set_logic_dc`
	`set_logic_one`
	`set_logic_zero`
	`set_opposite`
	`set_unconnected`

If you do not specify a driving cell, it is assumed during RTL budgeting that you are using the driver with the smallest area (which is often a weak driver). To override this behavior, you can specify a default driving cell by using the `set_default_driving_cell` command.

For more information about RTL budgeting, see the *Budgeting for Synthesis User Guide*.

Top-Down Environment Propagation. Table 2-2 lists the types of information propagated by top-down environment propagation and the commands used to define this information for the top-level design.

Note:

Top-down environment propagation propagates only information that was set from the top-level design.

Table 2-2 Top-Down Environment Propagation Details

Type of information	Constraint commands
Operating conditions	`set_operating_conditions`

Wire load models	`set_wire_load_min_block_size`
	`set_wire_load_mode`
	`set_wire_load_model`
	`set_wire_load_selection_group`

System interface	`set_driving_cell`

Design rule constraints	`set_max_capacitance`
	`set_max_fanout`
	`set_max_transition`
	`set_min_fanout`

Timing constraints	`create_clock`
	`group_path`
	`set_clock_gating_check`
	`set_clock_latency`
	`set_clock_transition`
	`set_clock_uncertainty`
	`set_critical_range`
	`set_false_path`
	`set_input_delay`
	`set_max_delay`
	`set_max_time_borrow`
	`set_min_delay`
	`set_multicycle_path`
	`set_output_delay`
	`set_propagated_clock`
	`set_resistance`
	`set_timing_ranges`

Table 2-2 Top-Down Environment Propagation Details (Continued)

Type of information	Constraint commands

Logic assignments	`set_case_analysis`
	`set_equal`
	`set_logic_dc`
	`set_logic_one`
	`set_logic_zero`
	`set_opposite`
	`set_unconnected`

Compile directives	`set_balance_registers`
	`set_boundary_optimization`
	`set_combinational_type`
	`set_compile_directives`
	`set_cost_priority`
	`set_dont_touch`
	`set_dont_touch_network`
	`set_fix_multiple_port_nets`
	`set_flatten`
	`set_implementation`
	`set_map_only`
	`set_minimize_tree_delay`
	`set_multibit_options`
	`set_resource_allocation`
	`set_register_type`
	`set_resource_implementation`
	`set_share_cse`
	`set_structure`
	`set_ultra_optimization`
	`set_ungroup`
	`set_wired_logic_disable`

Test attributes	`set_scan_configuration -style` `set_test_methodology`

Library cell	`set_compile_directive`
	`set_dont_touch`
	`set_dont_use`
	`set_prefer`

Design environment	`set_impl_priority`
	`set_local_link_library`
	`set_min_library`

In addition to the constraints and attributes specified in Table 2-2, top-down environment propagation also propagates the settings for any variables that have been set or modified in the current session.

Top-down environment propagation assigns an input delay of 30 percent of the related clock period for all partition input pins that do not have a default input delay. You can define a default input delay by using the `set_default_input_delay` command.

Top-down environment propagation assigns an output delay of 30 percent of the related clock period for all partition output pins that do not have a default output delay. You can define a default output delay by using the `set_default_output_delay` command.

For more information about these commands, see the man pages.

Generating the Compile Scripts

By default, the `acs_compile_design` command creates compile scripts for the top-level design and for each compile partition. For information about compile partitions, see “Identifying Compile Partitions” on page 2-18.

The compile scripts generated by Automated Chip Synthesis automatically check for errors and stop script execution when certain errors occur. For information about the errors that stop script execution, see “Script Termination Errors” on page 2-28.

In addition, the default compile script for the top-level design generates the following reports:

- Timing
- Area
- Constraints
- Quality of results

Automated Chip Synthesis places the generated script files in the $acs_work_dir/*dest_dir*/scripts directory (where *dest_dir* is the name of the destination directory).

Note:

For information about customizing the compile scripts, see "Generating the Compile Scripts" on page 3-24.

Automated Chip Synthesis generates the compile scripts when you run the `acs_compile_design` command from within Design Compiler.

The default partition compile script (shown in Example 2-1) performs a medium-effort full compile on each compile partition. Before performing the full compile, Automated Chip Synthesis places a `dont_touch` attribute on the subpartitions of the partition being compiled (if any exist). After the full compile, Automated Chip Synthesis saves the design (including any nonpartition subdesigns) in .ddc format (.db format if you are running in DB mode).

Example 2-1 XG Mode Default Partition Compile Script (acs_compile_design)

```
# read design
read_file -format ddc pass0/db/pre_compile/partition.ddc
current_design partition
link

# define verification setup file
set_svf pass0/svf/partition.svf

# apply constraints
set_dont_touch [get_designs subpartitions] true
source pass0/constraints/partition.con
set_max_area 0.0
set_flatten false

# full compile
compile -map_effort medium

# save .ddc file
write -format ddc -output pass0/db/post_compile/partition.ddc \
   {partition subdesigns}
```

The default top-level compile script (shown in Example 2-2) performs a medium-effort full compile on the top-level partition, followed by a top-level boundary compile. Before performing the full compile, Automated Chip Synthesis places a `dont_touch` attribute on all subpartitions. Before starting the boundary compile, Automated Chip Synthesis removes the `dont_touch` attributes (except on the reference designs for multiple instances). After the boundary compile, Automated Chip Synthesis saves the top-level partition and each subpartition in .ddc format (.db format if you are running in DB mode). For each compile partition, the partition design and its nonpartition subdesigns are saved. After saving the designs, Automated Chip Synthesis generates reports for the top-level design.

Example 2-2 XG Mode Default Top-Level Compile Script (acs_compile_design)

```
# read design
read_file -format ddc pass0/db/pre_compile/top_partition.ddc
current_design top_partition
link

# define verification setup file
set_svf pass0/svf/top_partition.svf

# apply logical constraints
set_dont_touch [get_designs subpartitions] true
source pass0/constraints/top_partition.con
set_max_area 0.0
set_flatten false

# full compile
compile -map_effort medium

# boundary compile
remove_attribute [find design {subpartitions}] dont_touch
set compile_top_acs_partition true
set compile_top_all_paths true
compile -top

# save .ddc files
write -format ddc -output pass0/db/post_compile/subpartition.ddc \
   {subpartition subdesigns}
write -format ddc -output pass0/db/post_compile/top_partition.ddc \
   {top_partition subdesigns}

# generate top-level reports
report_timing  > pass0/reports/RISC_CORE.tim
report_area  > pass0/reports/RISC_CORE.area
report_constraints  > pass0/reports/RISC_CORE.cstr
report_qor  > pass0/reports/RISC_CORE.qor

acs_write_html top_partition -destination pass0 -acs_work_dir $acs_work_dir
```

Script Termination Errors

Table 2-3 shows the errors that, by default, cause the script execution to stop. You can modify this list by changing the value of the `check_error_list` variable in your project .synopsys_dc.setup file.

Note:

For details about specific error codes, use the `man` command. For example, for details about the CMD-004 error, enter `man CMD-004`.

Table 2-3 Default Script Termination Errors

Description	Error codes
Licensing error occurred	UID-20, UIO-25, UIO-65, UIO-66
Invalid command syntax identified	CMD-004, CMD-006, CMD-007, CMD-008, CMD-009, CMD-010, CMD-011, CMD-012, CMD-014, CMD-019, CMD-026, CMD-031, CMD-037, DCSH-11, UI-11, UI-14, UI-15, UI-16, UI-17, UI-19, UID-32, UID-444, UIO-2, UIO-4, UIO-95, EQN-16, EQN-18
File I/O error occurred	CMD-015, CMD-016, DB-1, FILE-1, FILE-2, FILE-3, FILE-4, LNK-023, OPT-181, UI-20, UI-22, UI-23, UI-40, UID-9, UID-25, UID-28, UID-29, UID-30, UID-58, UID-270, UID-440, EQN-11
Problems with design specification	DES-001, UID-4, UID-13, UID-14, UID-19, UID-27, UID-272, UIO-94
Problems with design database	LINK-5, LINK-7, LINT-7, LINT-20, OPT-124, OPT-157, OPT-462, UID-103, UID-403, UIO-75
Problem with object specification	UID-109

Table 2-3 Default Script Termination Errors (Continued)

Description	Error codes
Problems with the technology libraries	OPT-101, OPT-102, OPT-114, OPT-128, OPT-155, UID-6, UID-7, UID-8, UID-15, UID-87, UIO-3
Problems with the synthetic libraries	OPT-127
Error occurred when running a script file	UI-21, UI-41
Arithmetic errors	EQN-15, EQN-20
Internal errors	EQN-6, OPT-100

Generating the Makefile

A makefile is a special script that defines the commands and dependencies required to build a target (in this case, compile a design). You automatically build your target by running a make utility. The make utility can be one of the make utilities supplied with UNIX or one supplied by a third-party program. By default, Automated Chip Synthesis uses gmake as the make utility.

Automated Chip Synthesis automatically generates a makefile to compile your design and places it in the destination directory ($acs_work_dir/*dest_dir*). The generated makefile is designed to run on the UNIX gmake utility.

If gmake is not installed on your system, you can download it from the following Web sites:

- www.gnu.org

  This Web site provides source code for the gmake utility.

- sunfreeware.com

  This Web site provides a compiled version of gmake for Solaris machines.

- hpux.cs.utah.edu

  This Web site provides a compiled version of gmake for HP-UX machines.

Caution!

If you are performing parallel compiles on multiple machines, a problem can occur if the system clocks on these machines are not synchronized (or if you are running on the HP-UX operating system). The gmake utility issues a message that "modification is in the future." If you are running gmake version 3.76 or greater, you can ignore this message (the compile job continues). If you are running an earlier version of gmake, the compile job might terminate and you will have to rerun the compile job.

Running the Compile Job

Using the generated makefile, Automated Chip Synthesis performs a bottom-up compile of your design. If your configuration supports it, Automated Chip Synthesis runs the partition compile jobs in parallel. Depending on which flow you are using, the compile process requires a Design-Compiler license for the duration of the compile job, plus an additional license for each active compile job.

Note:

To limit the number of licenses used by Automated Chip Synthesis, limit the number of parallel compile jobs, as described in “Specifying the Number of Parallel Compile Jobs” on page 3-42.

The `acs_compile_design` command writes the compiled designs (top-level design and the compile partitions) to the $acs_work_dir/*dest_dir*/db/post_compile directory.

Analyzing the Results

After running the compile job, run the `report_pass_data` command to determine whether the run finished successfully. The `report_pass_data` command checks the data directories for the precompile design database files, the compile scripts, and the postcompile design database files. If all of these files exist, the run was successful. If one or more files are missing, the run was not successful.

Analyzing a Successful Run

If your compile run finished successfully, analyze the design results by viewing the Hypertext Markup Language (HTML) summary file ($acs_work_dir/ACS_results_*design*.html, where *design* is the top-level design). This file is generated by the compile job (see "Using the HTML Report Files" on page 2-32 for more information about the HTML files).

If the analysis shows that the design meets constraints, you are done. If the design does not meet constraints, you can use the `acs_refine_design` command to refine the design. You might also want to customize aspects of the Automated Chip Synthesis flow. Chapter 3, "Customizing Automated Chip Synthesis," provides details about customization options.

Analyzing an Unsuccessful Run

If the compile run does not finish successfully, an HTML summary file is not generated. In this case, you can generate the HTML file by running the `acs_write_html` command and then use the information to analyze why the compile run failed. For information about errors that cause the compile script to terminate, see “Script Termination Errors” on page 2-28.

To continue, fix the errors and rerun the chip-level compile command.

Using the HTML Report Files

When you run a chip-level compile command, Automated Chip Synthesis creates an HTML summary file that contains information about the run. Each run is identified by its destination directory. You can access the HTML summary files through the top-level HTML results file ($acs_work_dir/ACS_results_*design*.html, where *design* is the top-level design), which contains a link to all HTML summary files. Figure 2-3 shows the top-level HTML results file after running pass0.

Figure 2-3 Top-Level HTML Results File

The HTML summary file includes the following information for each run:

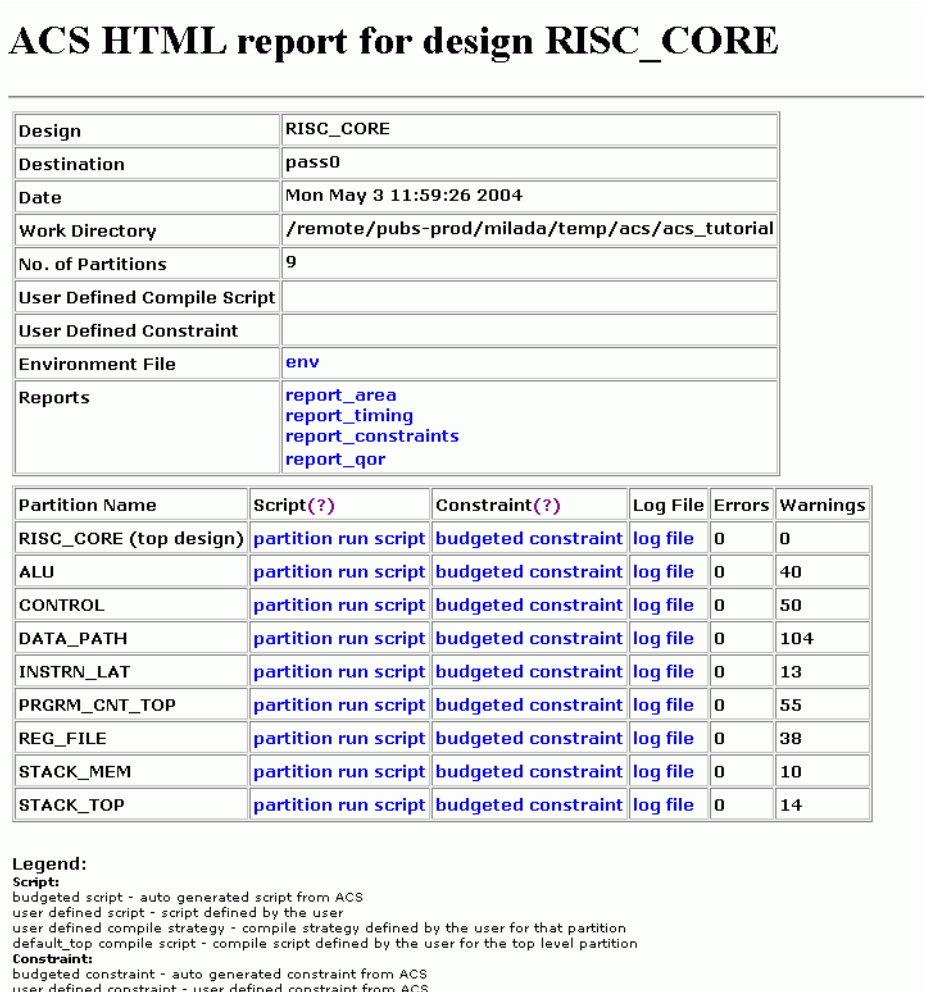
- The top-level design
- The destination directory name and location
- The time stamp
- The number of compile partitions
- Links to the user-defined scripts for that run
- A link to the environment file
- Links to the top-level reports generated during the compile run

In addition, the HTML summary file includes the following information for each compile partition:

- The partition name
- A link to the compile scripts (both the generated compile script and the user-defined compile script, if it exists)
- A link to the constraint files (both the generated constraint file and the user-defined constraint file, if it exists)
- A link to the compile log file
- The number of warnings generated during processing
- The number of errors that occurred during processing

Figure 2-4 shows an example of an HTML summary file.

Figure 2-4 HTML Summary File



ACS HTML report for design RISC_CORE

Design	RISC_CORE
Destination	pass0
Date	Mon May 3 11:59:26 2004
Work Directory	/remote/pubs-prod/milada/temp/acs/acs_tutorial
No. of Partitions	9
User Defined Compile Script	
User Defined Constraint	
Environment File	env
Reports	report_area report_timing report_constraints report_qor

Partition Name	Script(?)	Constraint(?)	Log File	Errors	Warnings
RISC_CORE (top design)	partition run script	budgeted constraint	log file	0	0
ALU	partition run script	budgeted constraint	log file	0	40
CONTROL	partition run script	budgeted constraint	log file	0	50
DATA_PATH	partition run script	budgeted constraint	log file	0	104
INSTRN_LAT	partition run script	budgeted constraint	log file	0	13
PRGRM_CNT_TOP	partition run script	budgeted constraint	log file	0	55
REG_FILE	partition run script	budgeted constraint	log file	0	38
STACK_MEM	partition run script	budgeted constraint	log file	0	10
STACK_TOP	partition run script	budgeted constraint	log file	0	14

Legend:
Script:
budgeted script - auto generated script from ACS
user defined script - script defined by the user
user defined compile strategy - compile strategy defined by the user for that partition
default_top compile script - compile script defined by the user for the top level partition
Constraint:
budgeted constraint - auto generated constraint from ACS
user defined constraint - user defined constraint from ACS

Refining Your Design

If your design does not meet constraints, use the `acs_refine_design` command to refine the design.

```
dc_shell-xg-t> acs_refine_design design
```

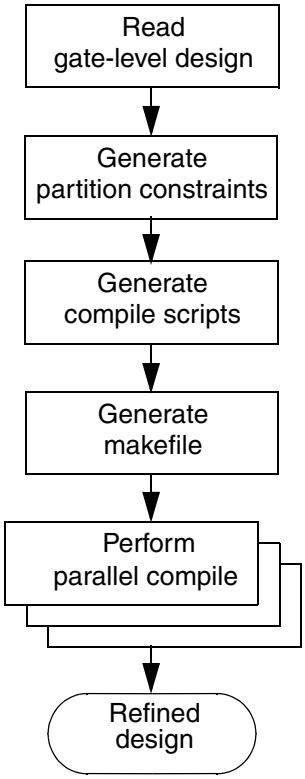
The `acs_refine_design` command uses the results of a previous compile run to update the design budget and refine the design. The default Automated Chip Synthesis flow uses the $acs_work_dir/pass0 directory as the source for updating the design budget and the $acs_work_dir/pass1 directory as the destination for the compile data. You can change the source directory by specifying the `-source` *`source_dir`* option when you invoke the `acs_refine_design` command. You can change the destination directory by specifying the `-destination` *`dest_dir`* option when you invoke the `acs_refine_design` command.

The `acs_refine_design` command performs the tasks shown in Figure 2-5.

Note:

If a makefile exists in the destination directory ($acs_work_dir/*dest_dir*/Makefile), the `acs_refine_design` command performs only the parallel compile task (unless you specify the `-force` option).

Figure 2-5 Refine Design Flow



The following sections describe these tasks.

Reading the Gate-Level Design

The `acs_refine_design` command first checks for the specified design in memory. If it does not locate the design in memory, it reads the postcompile design from the source directory—$acs_work_dir/*source_dir*/db/post_compile. If you are running in XG mode, the `acs_refine_design` command looks for the *design*.ddc file. If you are running in DB mode, the `acs_refine_design` command looks for the *design*.db file.

Note:

For information about changing the default location for the generated design database files, see "Customizing the Directory Structure" on page 3-2.

Generating Partition Constraints for Gate-Level Designs

Automated Chip Synthesis uses design budgeting to generate constraints for the gate-level compile partitions. Gate-level budgeting allocates timing and environment constraints among the compile partitions by taking into account the portion of the total slack required by each partition.

When generating partition constraints, the `acs_refine_design` command performs the following tasks:

1. Reads the design database files located in the $acs_work_dir/*source_dir*/db/post_compile directory (where *source_dir* is the name of the source directory)
2. Sets the current design to the top-level design
3. Sets the operating conditions

4. Allocates budgets for the subdesigns by using the `dc_allocate_budgets` command.

   If you need to perform preprocessing tasks before allocating the budgets or postprocessing tasks after allocating the budgets you can provide a custom budgeting script. For information about providing a custom budgeting script, see "Writing a Custom Budgeting Script" on page 3-22.

5. Generates constraint files for each compile partition based on the partition's budget

6. Writes the generated constraints to $acs_work_dir/*dest_dir*/constraints/*partition*.con (where *dest_dir* is the name of the destination directory)

Table 2-4 lists the types of constraints generated for each partition.

Table 2-4 Design Budgeting Details

Type of information	Constraint commands
Operating conditions	`set_operating_conditions`

Wire load models	`set_wire_load_min_block_size`
	`set_wire_load_mode`
	`set_wire_load_model`
	`set_wire_load_selection_group`

System interface	`set_drive`
	`set_driving_cell`
	`set_fanout_load`
	`set_load`

Design rule constraints	`set_max_capacitance`
	`set_max_fanout`
	`set_max_transition`
	`set_min_capacitance`
	`set_min_fanout`

Table 2-4 Design Budgeting Details (Continued)

Type of information	Constraint commands

Timing constraints	`create_clock`
	`group_path`
	`set_clock_gating_check`
	`set_clock_latency`
	`set_clock_transition`
	`set_clock_uncertainty`
	`set_critical_range`
	`set_false_path`
	`set_input_delay`
	`set_max_delay`
	`set_max_time_borrow`
	`set_min_delay`
	`set_multicycle_path`
	`set_output_delay`
	`set_propagated_clock`
	`set_resistance`
	`set_timing_ranges`

Logic assignments	`set_equal`
	`set_logic_dc`
	`set_logic_one`
	`set_logic_zero`
	`set_opposite`
	`set_unconnected`

If you do not specify a driving cell, it is assumed during design budgeting that you are using the driver with the smallest area (which is often a weak driver). To override this behavior, you can specify a default driving cell by using the `set_default_driving_cell` command.

For more information about design budgeting, see the *Budgeting for Synthesis User Guide*.

Generating the Compile Scripts

By default, the `acs_refine_design` command creates compile scripts for the top-level design and for each compile partition. For information about compile partitions, see “Identifying Compile Partitions” on page 2-18.

Automated Chip Synthesis places the generated script files in the $acs_work_dir/*dest_dir*/scripts directory (where *dest_dir* is the name of the destination directory).

Note:

For information about customizing the compile scripts, see “Generating the Compile Scripts” on page 3-24.

The default partition compile script (shown in Example 2-3) performs a high-effort incremental compile on each compile partition. Before performing the incremental compile, Automated Chip Synthesis places a `dont_touch` attribute on each subpartition of the design being compiled. Automated Chip Synthesis removes the `dont_touch` attributes (except on the reference designs for multiple instances) after completing the incremental compile.

Example 2-3 XG Mode Default Partition Compile Script (acs_refine_design)

```
# read design
read_file -format ddc pass1/db/pre_compile/partition.ddc
current_design partition
link

# define verification setup file
set_svf pass1/svf/partition.svf

# apply constraints
set_dont_touch [get_designs subpartitions] true
source pass1/constraints/partition.con
set_max_area 0.0

# incremental compile
compile -incr -map_effort high

# save .ddc file
write -format ddc -output pass1/db/post_compile/partition.ddc \
   {partition subdesigns}
```

The default top-level compile script (shown in Example 2-4) performs a top-level boundary compile on the top-level design.

Example 2-4 Default Top-Level Compile Script (acs_refine_design)

```
# read design
read_file -format ddc pass1/db/pre_compile/top_partition.ddc
current_design top_partition
link

# define verification setup file
set_svf pass1/svf/top_partition.svf

# apply logical constraints
source pass1/constraints/top_partition.con

# boundary compile
set compile_top_acs_partition true
set compile_top_all_paths true
compile -top

# save .ddc files
write -format ddc -output pass1/db/post_compile/subpartition.ddc \
   {subpartition subdesigns}
write -format ddc -output pass1/db/post_compile/top_partition.ddc \
   {top_partition subdesigns}

# generate top-level reports
report_timing  > pass1/reports/RISC_CORE.tim
report_area  > pass1/reports/RISC_CORE.area
report_constraints  > pass1/reports/RISC_CORE.cstr
report_qor  > pass1/reports/RISC_CORE.qor

acs_write_html top_partition -destination pass1 -acs_work_dir $acs_work_dir
```

Generating the Makefile

A makefile is a special script that defines the commands and dependencies required to build a target (in this case, compile a design). You automatically build your target by running a make utility. The make utility can be one of the make utilities supplied with UNIX or one supplied by a third-party program. By default, Automated Chip Synthesis uses gmake as the make utility.

Automated Chip Synthesis automatically generates a makefile to compile your design and places it in the destination directory ($acs_work_dir/*dest_dir*). The generated makefile is designed to run on the UNIX gmake utility.

Running the Compile Job

Using the generated makefile, Automated Chip Synthesis performs a bottom-up compile of your design. If your configuration supports it, Automated Chip Synthesis runs the partition compile jobs in parallel. The compile process requires a Design-Compiler license for the duration of the compile job, plus an additional license for each active compile job.

Note:

To limit the number of licenses used by Automated Chip Synthesis, limit the number of parallel compile jobs, as described in “Specifying the Number of Parallel Compile Jobs” on page 3-42.

The `acs_refine_design` command writes the optimized designs (top-level design and the compile partitions) to the $acs_work_dir/*dest_dir*/db/post_compile directory (where *dest_dir* is the name of the destination directory).

3

Customizing Automated Chip Synthesis

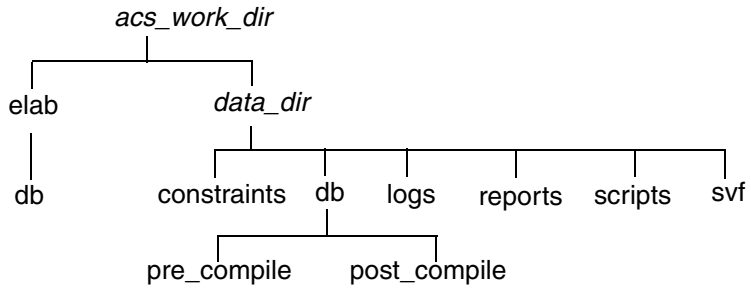
The default flow successfully optimizes most designs. However, in those cases where the default flow is not sufficient, Automated Chip Synthesis provides the capability to customize both the execution of tasks in the default flow and the flow itself.

This chapter describes the customization options in the following sections:

Customizing the Directory Structure

Figure 3-1 shows the default directory structure, where the top-level project directory is defined by the `acs_work_dir` variable. For information about the `acs_work_dir` variable, see "Invoking the Synthesis Tool" on page 2-2.

Figure 3-1 Default Directory Structure



To customize the directory structure, specify the file locations that differ from the default file locations by setting the `acs_user_dir` variable. To reset the directory structure to the default directory structure, use the `acs_reset_directory_structure` command.

Caution!

You must customize the directory structure (either in the .synopsys_dc.setup file or interactively) before you run the first chip-level compile command in a shell session. Automated Chip Synthesis ignores any changes you make to the directory structure after you have run a chip-level compile command.

The `acs_user_dir` variable is a Tcl array. For each element in the array, the element name is the path specification, and the element value is a list of file types. Automated Chip Synthesis supports both absolute and relative path specifications. Relative path specifications are relative to $acs_work_dir.

The syntax to specify the path location for a file type is

```
set acs_user_dir(path) file_type
```

Important:

The *path* argument cannot contain the @ character, except as part of the reserved word @pass@. For more information about the @pass@ reserved word, see “Specifying Locations for Pass-Dependent Files” on page 3-5.

Table 3-1 shows the file types defined by Automated Chip Synthesis and their default locations.

Table 3-1 Automated Chip Synthesis File Types

File type	Keyword	Default file location[1]
User-defined constraint file	user_override_constraint	$acs_work_dir/constraints/*dest_dir*
User-defined compile strategy	user_compile_strategy_script	$acs_work_dir/scripts/*dest_dir*
User-defined report script	user_report_script	$acs_work_dir/scripts/*dest_dir*
User-defined compile script	user_override_script	$acs_work_dir/scripts/*dest_dir*
User-defined budgeting script	user_budget_script	$acs_work_dir/scripts/*dest_dir*
Derived constraint file	pass_cstr	$asc_work_dir/*dest_dir*/constraints
Derived environment file	env	$asc_work_dir/*dest_dir*/constraints
Operating conditions for budgeting	oc.tcl	$asc_work_dir/*dest_dir*/constraints

Table 3-1 Automated Chip Synthesis File Types (Continued)

File type	Keyword	Default file location[1]
Constraints for budgeting	cstr.tcl	$asc_work_dir/*dest_dir*/constraints
Generated budgeting script	pass_budget_script	$acs_work_dir/*dest_dir*/scripts
Budgeting output log	pass_budget_output	$acs_work_dir/*dest_dir*/logs
Generated compile script	pass_compile_script	$acs_work_dir/*dest_dir*/scripts
Elaborated .ddc file	elab_ddc	$acs_work_dir/elab/db
Precompile .ddc file	pre_ddc	$acs_work_dir/*dest_dir*/db/pre_compile[1]
Postcompile .ddc file	post_ddc	$acs_work_dir/*dest_dir*/db/post_compile[1]
Elaborated .db file	elab_db	$acs_work_dir/elab/db
Precompile .db file	pre_db	$acs_work_dir/*dest_dir*/db/pre_compile
Postcompile .db file	post_db	$acs_work_dir/*dest_dir*/db/post_compile
Compile process makefile	makefile	$acs_work_dir/*dest_dir*
Compile log	pass_log	$acs_work_dir/*dest_dir*/logs
Verification setup file	svf	$acs_work_dir/*dest_dir*/svf
Area report	pass_area_report	$acs_work_dir/*dest_dir*/reports
Timing report	pass_timing_report	$acs_work_dir/*dest_dir*/reports
Constraint report	pass_cstr_report	$acs_work_dir/*dest_dir*/reports

1. The dest_dir argument refers to the destination directory.

Specifying Locations for Pass-Dependent Files

For all file types except the elaborated design database files (elab_db or elab_ddc file type), Automated Chip Synthesis creates or uses different versions of the files each time you run a chip-level compile command. These files are called pass-dependent files.

Automated Chip Synthesis uses a special notation to refer to the destination directory when defining the file locations. To use the destination directory in a path specification, use the reserved word `@pass@`.

For example, to place the precompile .ddc files in the destination directory, add the following command to your .synopsys_dc.setup file:

```
set acs_user_dir($acs_work_dir/@pass@/pre_compile) "pre_ddc"
```

If you do not use the `@pass@` variable in the path specification for pass-dependent files, Automated Chip Synthesis appends the destination directory name to the file name. For example, if you specify that all precompile .ddc files are placed in the $acs_work_dir/pre_compile directory, the file suffix becomes ddc.*dest_dir*.

Placing Multiple Files in the Same Location

If you want multiple file types to be in the same location, either specify all file types by using a single `set` command, or use the `lappend` command when specifying additional file types. If you use multiple `set` commands to define file types with the same location, you overwrite the existing information, resulting in an invalid directory structure.

For example, add either of the following command sequences to your .synopsys_dc.setup file to set the path location for the .ddc files as $acs_work_dir/ddc:

```
set acs_user_dir($acs_work_dir/ddc) "pre_ddc post_ddc"
```

or

```
set acs_user_dir($acs_work_dir/ddc) "pre_ddc"
lappend acs_user_dir($acs_work_dir/ddc) "post_ddc"
```

The following command sequence results in an invalid directory structure, because the path location for the pre_ddc file type is ignored.

```
set acs_user_dir($acs_work_dir/ddc) "pre_ddc"
set acs_user_dir($acs_work_dir/ddc) "post_ddc"
```

If you place multiple files with the same suffix in the same location, Automated Chip Synthesis appends the file type to the file name. For example, if you place both precompile and postcompile .ddc files in the same directory, the file suffix becomes .ddc.pre_ddc for the precompile files and .ddc.post_ddc for the postcompile files.

Accessing Files in a Customized Directory Structure

Automated Chip Synthesis provides the following capabilities to access files in the directory structure. It can

- Create the specified directory structure
- Report the specified directory structure
- Report the path specifications for specific file types
- Locate a specified file

The following sections describe these capabilities.

Creating the Directory Structure

Use the `acs_create_directories` command to create the directory structure. This command creates all directories in the directory structure with the exception of the data directories (these are created automatically when you run a chip-level compile command).

```
dc_shell-xg-t> acs_create_directories
```

Note:

If you want to create a data directory manually, use the `create_pass_directories` command.

Reporting the Directory Structure

Use the `acs_report_directories` command to report the current directory structure. This command lists (in alphabetical order) each directory defined in the current directory structure and the types of files located in each directory.

```
dc_shell-xg-t> acs_report_directories
```

Each line in the generated report has the following format:

directory : {*file_types*}

Reporting Path Specifications

To determine the path specifications for specific file types, use the `acs_report_directories -file_types` command. This command reports the path specification for the specified file types in the format defined earlier. For example, to report the path specification for the elab_ddc and pre_ddc file types, enter the following command:

```
dc_shell-xg-t> acs_report_directories \
   -file_types {elab_ddc pre_ddc}
Report of ACS directory structure
elab_ddc -> /usr/designs/my_design/elab/db
pre_ddc -> /usr/designs/my_design/@pass@/pre_db
```

Locating Files

To return the location of a specific file, use the `acs_get_path` command. To return the location of a pass-dependent file type, you must specify the data directory as well as the file type.

For example, to get the location of the elab_ddc file, enter

```
dc_shell-xg-t> acs_get_path -file_type elab_ddc
/usr/designs/my_design/elab/db
```

To get the location of the pass0 pre_ddc file, enter

```
dc_shell-xg-t> acs_get_path -file_type pre_ddc -pass pass0
/usr/designs/my_design/pass0/db/pre_compile
```

Unlike with the `acs_report_directories` command, you can use the `acs_get_path` command as input to another command. When you are customizing scripts or chip-level compile commands (as described later in this chapter), use the `acs_get_path` command to specify file locations. By using the `acs_get_path` command instead of hardcoding the locations, you ensure support of customized directory structures.

Controlling Naming Conventions

Automated Chip Synthesis provides variables that control the naming conventions for the data directories and the files used or generated by the following tasks:

- Top-down environment propagation (generated constraint files)
- Design budgeting (script files, budgeting results, and generated constraint files)
- Compile process (compile scripts, makefile, and compile results)

Caution!

If you are using a customized directory structure and want to use nondefault file naming conventions, you must set the variables that control the file naming conventions in the

.synopsys_dc.setup file. Set these variables *before* you define the customized directory structure (by setting the `acs_user_dir` variable). If you change the file naming conventions after defining the customized directory structure, problems can occur.

Table 3-2 lists these variables by task. Some variables control the complete file name, while others control only the file name suffix. If the variable controls only the file name suffix, the complete file name is *partition*.*suffix*.

Table 3-2 Variables to Control File Naming Conventions

Variable	Default value	Description
Data Directories		
`acs_default_pass_name`	pass	Prefix for the default data directory names. The pass number (either 0 or 1) is added to the prefix to generate the directory name.
Top-Down Environment Propagation		
`acs_constraint_file_suffix`	con	Suffix for the constraint files generated from propagated constraints
Formality Interface		
`acs_svf_suffix`	svf	Suffix for the generated SVF files
Design Budgeting		
`acs_user_budgeting_script`	budget.scr	Name of the user-defined budgeting script

Table 3-2 Variables to Control File Naming Conventions (Continued)

Variable	Default value	Description
Compile Process		
acs_area_report_suffix	area	Suffix for the area report files
acs_compile_script_suffix	autoscr	Suffix for the generated compile scripts
acs_cstr_report_suffix	cstr	Suffix for the constraint report files
acs_db_suffix	db	Suffix for the generated .db files
acs_global_user_compile_ strategy_script	default	Base name of the user-defined compile strategy used for all partitions
acs_log_file_suffix	log	Suffix for the compile log files
acs_makefile_name	Makefile	Name of the generated makefile
acs_override_report_suffix	report	Suffix for a user-defined report script
acs_override_script_suffix	scr	Suffix for the user-defined compile script used for a specific partition
acs_timing_report_suffix	tim	Suffix for the timing report files
acs_user_compile_strategy_ script_suffix	compile	Suffix for a user-defined compile strategy
acs_user_top_compile_ strategy_script	default_top	Base name of the user-defined compile strategy for the top-level partition

Customizing Tasks in the Default Flow

You can customize the following tasks in the default flow:

- Generating the makefile
- Resolving multiple instances
- Identifying compile partitions
- Generating partition constraints
- Generating compile scripts
- Running the compile job

The following sections describe these options.

Generating the Makefile

Automated Chip Synthesis generates a makefile to compile your design. Because the makefile contents vary depending on how the compile jobs are run, you must select the compile method before using Automated Chip Synthesis.

Automated Chip Synthesis supports the following methods:

- Invoking the compile shell directly from the make utility

  By default, the makefile generated by Automated Chip Synthesis invokes the compile shell directly from the make utility.

- Running the compile job in batch mode

  To run the compile job in batch mode, use the `acs_submit` and `acs_submit_large` commands to enable batch submission. For more information about using the `acs_submit` and `acs_submit_large` commands, see "Running the Compile Job in Batch Mode" on page 3-44.

Resolving Multiple Instances

Before you compile your design, all multiple instances in the design must be resolved.

To check your design for multiple instances, use the following command:

```
dc_shell-xg-t> sub_designs_of -multiple_instances \
   -hierarchy design
```

By default, Automated Chip Synthesis resolves multiple instances by selecting a default master instance for each set of instances having the same reference design (see "Resolving Multiple Instances" on page 2-17).

Caution!

Neither Automated Chip Synthesis nor the `uniquify` command support multiple instances of black box cells. You must manually correct this problem before running Automated Chip Synthesis.

After identifying the compile partitions (by running either the `acs_compile_design` command or the `set_compile_partitions` command), you can report the default master instance by using the `get_cells` command.

```
dc_shell-xg-t> get_cells -hierarchical \
   -filter "@masterInstance_soft == true"
```

For more information about the `acs_compile_design` command, see “Compiling Your Design” on page 2-15. For more information about the `set_compile_partitions` command, see “Specifying Compile Partitions” on page 3-18.

If your design contains multiple instances and the default master instance is not sufficient, use one of the following methods to resolve the multiple instances:

- Select a master instance.
- Uniquify the design.
- Ungroup the subdesigns containing the instances.

Caution!

In some cases, using a master instance can reduce the quality of results. If the multiple instances are very different (for example, constant logic drives one or more input ports or the instances use different clocks), resolve the multiple instances by using the uniquify or ungroup methods.

The following sections describe these methods.

Selecting a Master Instance

Use the `set_attribute` command to set the `MasterInstance` attribute on the instance you want to use to represent all of the multiple instances.

```
dc_shell-xg-t> set_attribute -type boolean instance \
   MasterInstance true
```

This method is similar to using the compile-once-don't-touch method on a mapped design. Only the instance with the `MasterInstance` attribute is compiled.

Uniquifying the Design

Use the `uniquify` command on the top-level design to ensure that all design instances are unique. In rare instances, the `uniquify` command leaves nonunique design instances. To prevent this from occurring, use the `-force` option on the `uniquify` command.

```
dc_shell-xg-t> uniquify -force
```

Ungrouping a Subdesign

Use the `ungroup` command on the subdesigns containing multiple instances.

```
dc_shell-xg-t> ungroup subdesign
```

Partitioning the Design

When you partition a design for Automated Chip Synthesis, you do not change the hierarchy of the design. You identify the subdesigns on which to perform a top-down compile during the chip compilation process.

By default, Automated Chip Synthesis uses the following subdesigns as the compile partitions:

- First-level subdesigns (hierarchical children of the top-level design)
- Reference designs of multiple instances

Instead of using the first-level subdesigns as the compile partitions, Automated Chip Synthesis can select the compile partitions based on subdesign size estimates. To enable this autopartitioning capability, set the `acs_use_autopartition` variable to true.

These subdesigns might not be the best set of compile partitions for your design. The following sections describe how you can set the compile partitions.

Partitioning Guidelines

When selecting compile partitions, consider the following issues:

- If you make the partitions very small, you might create artificial boundaries that restrict optimization.

  Use the following partitioning guidelines to minimize the quality-of-results impact:

  - Keep related combinational logic and its destination register together.

- Eliminate glue logic.
- Partition by design goal.
- Partition by compile technique.

  For more information about these partitioning guidelines, see the *Design Compiler User Guide*.

- If you make the partitions very big, partition compile runtimes can be lengthy.

  Consider the performance and memory capacity of the available CPUs when you determine the size of each compile partition.

Smart partitioning can reduce the total chip compile runtime. Use the following partitioning guidelines to minimize the total runtime:

- Consider the number of CPUs available for parallel compiles when you determine the number of compile partitions.
- Balance the number of partitions in each branch of the hierarchy.

  A parent partition (or the top-level design) cannot be compiled until all of its child partitions are compiled. If hierarchy branches have varying numbers of partitions, you might have CPUs that are idle while they wait for a child partition to compile.

- Balance the gate count in each partition.

  The compile time for a partition is related to the size of the partition. The compile time for a design depends on the longest cumulative partition compile time along a path in the design's hierarchy. By making the partition compile times comparable to each other, you shorten the overall design compile time.

Specifying Compile Partitions

Use the `set_compile_partitions` command to specify the compile partitions for the current design.

Caution!

If a subdesign has a `dont_touch` attribute, Automated Chip Synthesis will not use that subdesign as a compile partition.

In addition to the compile partitions you specify (or the size-based compile partitions in the case of autopartitioning), the reference designs for multiple instances also become compile partitions. If your design contains multiple instances and you want to use default master instances, specify the `-force` option. If you do not specify the `-force` option, the `set_compile_partitions` command fails if the design contains unresolved multiple instances.

You can partition the design in the following ways:

- If you know the design well, specify the subdesigns to use as partitions (`-designs` option).

  ```
  dc_shell-xg-t> set_compile_partitions -designs {A B C} \
     -force
  ```

- If you want to quickly make a first attempt at partitioning the design, use all subdesigns at a specified level of the hierarchy as partitions (`-level` option). Level 1 refers to the children of the top-level design.

  For example, to use all hierarchical children of the top-level design as partitions, enter the following command:

  ```
  dc_shell-xg-t> set_compile_partitions -level 1 -force
  ```

Important:

When you use the `-level` option, Automated Chip Synthesis uses only the subdesigns at the specified level as compile partitions. To create compile partitions at multiple levels, use the `-designs` option.

- If you want Automated Chip Synthesis to select compile partitions, run autopartitioning (`-auto` option).

  ```
  dc_shell-xg-t> set_compile_partitions -auto -force
  ```

  For more information about autopartitioning, see “About Autopartitioning” on page 3-20.

- Use all subdesigns as partitions `(-all` option)

  ```
  dc_shell-xg-t> set_compile_partitions -all -force
  ```

When you run the `set_compile_partitions` command, Automated Chip Synthesis sets the `is_partition` attribute to true on the following designs:

- The top-level design
- The subdesigns specified on the command line
- The reference designs of multiple instances

Automated Chip Synthesis uses the `is_partition` attribute to derive partition-level constraints, to generate compile scripts, and to generate the makefile for the chip-level compilation.

Use the `report_partitions` command to list the compile partitions in the current design, along with their relative size. For a sample partitions report, see Example 4-2 on page 4-10.

Note:

If the compile partitions have not yet been set, the `report_partitions` command shows the design hierarchy and marks the top-level design as a compile partition. However, no compile partitions actually exist until you run the `acs_compile_design` or `set_compile_partitions` command.

About Autopartitioning

When performing autopartitioning, Automated Chip Synthesis selects the compile partitions based on subdesign size and connectivity.

- For RTL designs, Automated Chip Synthesis estimates the subdesign size.
- For gate-level designs, Automated Chip Synthesis uses the cell area as the subdesign size.

By default, Automated Chip Synthesis tries to select compile partitions with a maximum size of approximately 10 percent of the total design size. Use the `report_partitions` command to list the compile partitions and their relative size.

You can change the maximum percentage value by setting the `acs_autopart_max_percent` variable. For a very large design, you might get better results by having more partitions that each compile faster, so you would reduce the maximum percentage value. For a small design, you might have enough compute power to have fewer compile partitions, so you would increase the maximum percentage value. For example,

```
dc_shell-xg-t> set acs_autopart_max_percent 25
dc_shell-xg-t> set_compile_partitions -auto -force
```

For a gate-level design, you can control autopartitioning by specifying a maximum cell area for a compile partition, instead of a maximum percentage. To specify the maximum cell area for autopartitioning, set the `acs_autopart_max_area` variable. For example,

```
dc_shell-xg-t> set acs_autopart_max_area 100000
dc_shell-xg-t> set_compile_partitions -auto -force
```

For RTL designs, the size estimates are relative and you should limit the partition size using the maximum percentage rather than the maximum area.

Note:

If you specify both `acs_autopart_max_percent` and `acs_autopart_max_area`, `acs_autopart_max_area` takes precedence.

Changing Compile Partitions

By default, the `set_compile_partitions` command removes all existing compile partitions before setting the specified partitions. When you want to add partitions to the existing partitions, you can override this behavior by specifying the `-no_reset` option.

You can also use the `remove_attribute` command to remove the partition settings.

```
dc_shell-xg-t> remove_attribute \
   [get_designs -filter "@is_partition==true"] is_partition
```

Generating Partition Constraints

Automated Chip Synthesis provides three methods of customizing the partition constraints:

- Writing a Custom Budgeting Script

  Gate-level budgeting supports the use of a custom budgeting script. RTL budgeting does not provide this capability.

- Using an Override Constraint File

  Instead of using constraints generated by Automated Chip Synthesis, you can provide an override constraints file.

These customization options are described in the following sections.

Writing a Custom Budgeting Script

When performing gate-level design budgeting, Automated Chip Synthesis checks whether a custom budgeting script (budget.scr) exists in the directory defined for the user_budget_script file type (default directory is $acs_work_dir/scripts/*dest_dir*). If the file exists, Automated Chip Synthesis runs the script instead of running the budgeting command directly.

Note:

You must generate a custom budgeting script for each chip-level compile run in which you want to override the default.

A custom budgeting script includes the following items:

- Preprocessing commands (optional)
- The appropriate budgeting command
- Postprocessing commands (optional)

For example, assume you want to scale the budgeted constraints to make them more aggressive. Example 3-1 shows a custom budgeting script that scales the budgeted constraints by 10 percent.

Example 3-1 Custom Budgeting Script

```
set instance_list \
   [sub_instances_of $design_name \
    -partition -hier -master -names]

# preprocessing command
set_context_margin -max -percent 10

# allocate_budgets command
dc_allocate_budgets -mode gate $instance_list
write_partition_constraints $design_name \
   -destination $this_pass -hier
```

Using an Override Constraint File

If the partition constraints generated by Automated Chip Synthesis do not meet your requirements, you can provide override constraint files.

Automated Chip Synthesis supports two types of override constraint files: default and design-specific. The default override constraint file (default.con) overrides the generated constraints for all partitions. The design-specific override constraint file (*partition*.con) applies only to the specific partition.

When generating a compile script, Automated Chip Synthesis checks whether override constraint files (*partition*.con or default.con) exist in the directory defined for the user_override_constraint file type (default directory is $acs_work_dir/constraints/*dest_dir*). If override constraint files exist, Automated Chip Synthesis reads the constraint files in the following order:

1. Generated constraint file
2. Default override constraint file (default.con)
3. Design-specific override constraint file (*partition*.con)

Note:

If a custom compile script exists in the user_override_script directory (default directory is $acs_work_dir/scripts/*dest_dir*), Automated Chip Synthesis uses the custom compile script, rather than the generated compile script, when compiling the particular partition. For more information about custom compile scripts, see "Writing a Custom Compile Script" on page 3-38.

You must provide override constraint files for each chip-level compile run in which you want to override the generated constraints.

Generating the Compile Scripts

Automated Chip Synthesis provides four methods of customizing the compile scripts:

- Setting Compile Attributes
- Writing a Custom Compile Strategy
- Writing a Custom Report Script
- Writing a Custom Compile Script

The following sections describe these methods.

Setting Compile Attributes

Automated Chip Synthesis provides compile attributes that you can use to control the `compile` command, the compile effort and the compile strategy of the compile scripts generated by the chip-level compile commands. This section describes the commands used to set and report on the compile attributes, as well as the supported compile attributes.

Overview of Automated Chip Synthesis Attributes. The Automated Chip Synthesis compile attributes differ from the general attributes used by the synthesis tools. Automated Chip Synthesis provides a set of commands for working with these attributes. You must use these commands, rather than the general attribute commands, when working with the compile attributes.

Note:

Before you can set compile attributes, your design must contain one or more compile partitions (subdesigns with the `is_partition` attribute set). For information about specifying compile partitions, see “Specifying Compile Partitions” on page 3-18.

You set compile attributes on a compile partition by using the `acs_set_attribute` command. For example,

```
dc_shell-xg-t> acs_set_attribute IncrementalCompile none
```

By default, the specified attribute is set on all compile partitions in the current design. To set the attribute only on specific compile partitions, use the `-partitions` option to specify the affected partitions. For example,

```
dc_shell-xg-t> acs_set_attribute -partitions A \
   IncrementalCompile none
```

To ensure that you achieve the desired change to the compile strategy, the `acs_set_attribute` command validates the compile attribute name and value and generates a warning message when it detects invalid values.

To see the compile attributes set on your design, use the `acs_report_attribute` command.

```
dc_shell-xg-t> acs_report_attribute
```

For more information about the `acs_set_attribute` and `acs_report_attribute` commands, see the man pages.

Changing the compile Command. Automated Chip Synthesis supports both the `compile` and `compile_ultra` commands. By default (the `CompileUltra` attribute is false), the chip-level compile commands use the `compile` command. To use the `compile_ultra` command instead, set the `CompileUltra` compile attribute to true.

Changing the Compile Effort. Automated Chip Synthesis compiles each partition using two compile steps:

1. The first compile step consists of either a full compile or an incremental compile, depending on which chip-level compile command you are running.

   - Full compile (when running `acs_compile_design` or `acs_recompile_design`)

     By default, Automated Chip Synthesis performs a full compile on all compile partitions, including the top-level partition.

   - Incremental compile (when running `acs_refine_design`)

     By default, Automated Chip Synthesis performs an incremental compile step only on the subpartitions, not on the top-level partition.

   Before performing the first compile step (whether it is a full compile or an incremental compile), Automated Chip Synthesis places a `dont_touch` attribute on each subpartition of the design being compiled. Automated Chip Synthesis removes the `dont_touch` attributes after completing this compile step.

2. Boundary compile

   By default, Automated Chip Synthesis performs this compile step only on the top-level compile partition.

If you are using the `compile` command (the `CompileUltra` attribute is false), you can control the effort level for each of these steps (or disable the compile step) by setting the appropriate compile effort attribute: `FullCompile`, `IncrementalCompile`, or `BoundaryCompile`.

Note:

The `FullCompile` attribute affects only the compile scripts generated by the `acs_compile_design` and `acs_recompile_design` commands. The `IncrementalCompile` attribute affects only the compile scripts generated by the `acs_refine_design` command.

Table 3-3 shows the valid values for each compile effort attribute. For each attribute value, the table also shows the resulting `compile` command.

Table 3-3 Compile Effort Attributes

Attribute	Valid values (default in bold)	Impact on compile script
`FullCompile` (attribute type = string)	none low **medium** high	none: Prevents the full compile step from occurring. `CompileUltra`=false and `FullCompile`<>none: `compile -map_effort` *`value`*
`IncrementalCompile` (attribute type = string)	none low medium **high**	none: Prevents incremental compile step from occurring. `CompileUltra`=false and `IncrementalCompile`<>none: `compile -incr -map_effort` *`value`*

`BoundaryCompile`
 	
(attribute type = string)
	none
	low
	medium
	high
	top
		none:
		Prevents the boundary compile step from occurring.
		
		`CompileUltra`=false and `BoundaryCompile`=low, medium, or high:
		`compile -incr -map_effort` *`value`*
		
		`CompileUltra`=false and `BoundaryCompile`=top:
		`set compile_top_acs_partition true`
		`compile -top`

Table 3-4 provides links to the default compile scripts for each chip-level compile command. In these scripts you can see how the default effort level for each compile step is implemented.

Table 3-4 Default Compile Scripts

Chip-Level compile command	Partition compile script	Top-Level compile script
`acs_compile_design`	Example 2-1 on page 2-26	Example 2-2 on page 2-27
`acs_refine_design`	Example 2-3 on page 2-41	Example 2-4 on page 2-42
`acs_recompile_design`	Example 3-6 on page 3-57	Example 3-7 on page 3-57

Changing the Compile Strategy. Automated Chip Synthesis provides attributes that you can set to control the compile strategy that is used during synthesis.

Table 3-5 on page 3-31 lists the attributes that modify the compile strategy for each compile step. The table shows the data type and valid values for each attribute (the default value is shown in bold text). In Table 3-5, the following keywords are used to represent the various compile steps: full (full compile), incr (incremental compile), and bnd (boundary compile). For information about the compile steps used by Automated Chip Synthesis, see "Changing the Compile Effort" on page 3-27. For information about compile strategies, see the Design Compiler documentation.

The attributes shown in Table 3-5 affect the compile scripts generated by the chip-level compile commands. Some attributes add commands to a script before the `compile` command; others modify the `compile` command options. Only attribute values that modify the default compile scripts are listed in the tables; attribute values that do not cause a change from the default scripts are not included.

Note:

If a custom compile strategy or a custom compile script applies to a particular partition, the Automated Chip Synthesis compile attributes will not affect the synthesis results for that partition.

If a custom compile strategy exists in the user_compile_strategy_script directory (the default directory is $acs_work_dir/scripts/*dest_dir*), Automated Chip Synthesis uses the custom compile strategy instead of generating a compile strategy. For more information about custom compile strategies, see “Writing a Custom Compile Strategy” on page 3-34.

If a custom compile script exists in the user_override_script directory (the default directory is $acs_work_dir/scripts/*dest_dir*), Automated Chip Synthesis uses the custom compile script, rather than the generated compile script, when compiling the particular partition. For more information about custom compile scripts, see “Writing a Custom Compile Script” on page 3-38.

Table 3-5 Compile Strategy Attributes

Attribute	Compile steps	Impact on compile script

`OptimizationPriorities`
(attribute type = string)
 	
Valid values:
• area
• area_timing
• timing
• **timing_area**
	full	area:
		`set_minimize_tree_delay false`
		`set_resource_allocation area_only`
		`set_resource_implementation area_only`
		`set compile_sequential_area_recovery true`
		#`CompileUltra`=false:
		`compile -area_effort high`
		#`CompileUltra`=true:
		`compile_ultra`
		
		area_timing:
		`set_resource_allocation area_only`
		`set_resource_implementation area_only`
		`set compile_sequential_area_recovery true`
		#`CompileUltra`=false:
		`compile -area_effort high`
		#`CompileUltra`=true:
		`compile_ultra`
		
		timing:
		(ignore maximum area constraint)
		
		timing_area:
		(set maximum area constraint)

`OptimizationPriorities`
(attribute type = string)
 	
Valid values:
• area
• area_timing
• timing
• **timing_area**
	incr	timing:
		(ignore maximum area constraint)
		
		timing_area:
		(set maximum area constraint)
		
		area, area_timing:
		`set_resource_allocation area_only`
		`set_resource_implementation area_only`
		`set compile_sequential_area_recovery true`

Table 3-5 Compile Strategy Attributes (Continued)

Attribute	Compile steps	Impact on compile script
`OptimizationPriorities` (attribute type = string) Valid values: • area • area_timing • timing • **timing_area**	bnd	timing: (ignore maximum area constraint) timing_area: (set maximum area constraint) area, area_timing: `set compile_sequential_area_recovery true`

`AutoUngroup`
(attribute type = string)
 	
(syntax of attribute value is "*`mode [numcells]`*")
 	
Valid *mode* values:
• **false**
• delay
• area
 	
Valid *numcells* values:
integer > 0
	full	false:
		#`CompileUltra`=false:
		`compile`
		#`CompileUltra`=true:
		`compile_ultra -no_auto_ungroup`
		
		"delay *numcells*":
		`set compile_auto_ungroup_delay_num_cells \`
		`  numcells`
		`set compile_auto_ungroup_hierarchy 0`
		#`CompileUltra`=false:
		`set_ultra_optimization -force`
		`compile -auto_ungroup mode`
		#`CompileUltra`=true:
		`compile_ultra`
		
		"area *numcells*":
		`set compile_auto_ungroup_area_num_cells \`
		`  numcells`
		`set compile_auto_ungroup_hierarchy 0`
		#`CompileUltra`=false:
		`set_ultra_optimization -force`
		`compile -auto_ungroup mode`
		#`CompileUltra`=true:
		`compile_ultra`

Table 3-5 Compile Strategy Attributes (Continued)

Attribute	Compile steps	Impact on compile script

`CanFlatten`
(attribute type = boolean)
 	
Valid values:
• true
• **false**
	full	true:
		`set_flatten true -effort medium \`
		`  -phase true -minimize multiple_output`
		
		false:
		`set_flatten false`

`CompileVerify`[1,2] (attribute type = boolean) Valid values: • true • **false**	full incr	`CompileVerify`=true and `CompileUltra`=false: `compile -verify`
`MaxArea` (attribute type = float) Valid values: *float* > 0.0 **(default is 0.0)**	full incr	`OptimizationPriorities` = area: `set_max_area` *`value`* `-ignore_tns` `OptimizationPriorities` = area_timing, `OptimizationPriorities` = timing_area: `set_max_area` *`value`*

`PreserveBoundaries`
(attribute type = boolean)
 	
Valid values:
• true (default when CompileUltra=false)
• false (default when CompileUltra=true)
	full
	incr
		`PreserveBoundaries`=false
		#`CompileUltra`=false:
		`  compile -boundary_optimization`
		#`CompileUltra`=true:
		`  compile_ultra`
		
		`PreserveBoundaries`=true
		#`CompileUltra`=false:
		`  compile`
		#`CompileUltra`=true:
		`  compile_ultra -no_boundary_optimization`

Table 3-5 Compile Strategy Attributes (Continued)

Attribute	Compile steps	Impact on compile script

`TestReadyCompile`
(attribute type = boolean)
 	
Valid values:
• true
• **false**
	full
	incr
		true:
		#`CompileUltra`=false:
		`compile -scan`
		#`CompileUltra`=true:
		`compile_ultra -scan`

`UltraOptimization`[1]
(attribute type = boolean)
 	
Valid values:
• true
• **false**
	full
	incr
	bnd
		`UltraOptimization`=true and `CompileUltra`=false:
		`set_ultra_optimization -force`
		`compile`

1. *This attribute is ignored if CompileUltra is true.*
2. *This attribute is not supported in XG mode.*

Writing a Custom Compile Strategy

If the compile strategy attributes do not provide for sufficient customization of the compile strategy, you can provide a custom compile strategy file.

Automated Chip Synthesis supports three types of custom compile strategies:

- Top-level partition

  This custom compile strategy overrides the compile strategy generated by Automated Chip Synthesis for the top-level partition.

The default name for this compile strategy file is default_top.compile. To use a different name for this file, set the `acs_user_top_compile_strategy_script` variable.

- Global compile partition

  This custom compile strategy overrides the compile strategy generated by Automated Chip Synthesis for all compile partitions (except the top-level partition).

  The default name for this compile strategy file is default.compile. To use a different name for this file, set the `acs_global_user_compile_strategy_script` variable.

- Specific compile partition

  This custom compile strategy script overrides all other compile strategies for a specific compile partition.

  The name for this custom compile strategy script is *partition*.compile.

When generating a compile script, Automated Chip Synthesis checks whether a custom compile strategy file (*partition*.compile, default_top.compile, or default.compile) exists in the directory defined for the user_compile_strategy_script file type (default directory is $acs_work_dir/scripts/*dest_dir*). If a custom compile strategy file exists, Automated Chip Synthesis uses it instead of generating a compile strategy.

Note:

If a custom compile script exists in the user_override_script directory (default directory is $acs_work_dir/scripts/*dest_dir*), Automated Chip Synthesis uses the custom compile script,

rather than the generated compile script, when compiling the particular partition. For more information about custom compile scripts, see “Writing a Custom Compile Script” on page 3-38.

A custom compile strategy file includes the following items:

- Compile variable settings (optional)
- Compile attribute settings (optional)
- The `compile` command

  You might have more than one compile command in your custom compile strategy file (for example, when you run both a full compile and a top-level boundary compile).

  Note:

  If your custom compile strategy performs a top-level boundary compile (`compile -top`), use the `acs_remove_dont_touch` command to remove the `dont_touch` attributes from the subpartitions (except on the reference designs for multiple instances) before starting the top-level boundary compile.

You must provide custom compile strategy files for each chip-level compile run in which you want to override the default.

Example 3-2 shows a custom compile strategy file.

Example 3-2 Custom Compile Strategy

```
# Set compile variables
set compile_new_boolean_structure false

# Set compile attributes
set_structure true -timing true -boolean false
set_max_area 0
set_flatten false

# Run the compile command
compile  -map_effort medium
```

Writing a Custom Report Script

If the default reports generated by Automated Chip Synthesis do not meet your requirements, you can provide a custom report script.

Automated Chip Synthesis supports two types of custom report scripts: default and design specific. The default custom report script (default.report) overrides the default report script for all partitions. The design-specific custom report script (*partition*.report) applies only to the specific partition and overrides the default report script.

When generating a compile script, Automated Chip Synthesis checks whether a custom report script (*partition*.report or default.report) exists in the directory defined for the user_override_script file type (default directory is $acs_work_dir/scripts/*dest_dir*). If a custom report script exists, Automated Chip Synthesis uses it instead of the default report script.

Note:

If a custom compile script exists in the user_override_script directory (default directory is $acs_work_dir/scripts/*dest_dir*), Automated Chip Synthesis uses the custom compile script,

rather than the generated compile script, when compiling the particular partition. For more information about custom compile scripts, see “Writing a Custom Compile Script” on page 3-38.

You must provide custom report scripts for each chip-level compile run in which you want to override the default.

Example 3-3 shows a custom report script.

Example 3-3 Custom Report Script

```
# Generate reports
set my_design $current_design
report_timing > \
   [acs_get_path -file_type pass_timing_report \
    -pass pass0 -name $my_design.tim]
report_area > \
   [acs_get_path -file_type pass_area_report \
    -pass pass0 -name $my_design.area]
report_constraints > \
   [acs_get_path -file_type pass_cstr_report \
    -pass pass0 -name $my_design.cstr]
```

Writing a Custom Compile Script

If the generated compile script does not meet your requirements, you can provide a custom compile script.

When compiling a design, Automated Chip Synthesis checks whether a custom compile script (*partition*.scr) exists in the directory defined for the user_override_script file type (default directory is $acs_work_dir/scripts/*dest_dir*). If the file exists, Automated Chip Synthesis uses it instead of the default compile script.

A custom compile script includes the following tasks:

- Sets the environment variables

  When you run a chip-level compile command, Automated Chip Synthesis captures the environment variable settings in a file called env. This file is located in the directory defined for the env file type (default directory is $acs_work_dir/*dest_dir*/scripts).

- Reads the design
- Links the design
- Sets the constraints
- Defines the compile strategy

  For more information about defining the compile strategy, see "Writing a Custom Compile Strategy" on page 3-34.

- Compiles the design
- Writes the design to the destination directory

  You must write the design to the directory defined for the post_ddc file type with the name *design*.ddc. If you are running in DB mode, you must write the design to the directory defined for the post_db file type with the name *design*.db.

- Generates reports (optional)
- Performs error checking (optional)

  Automated Chip Synthesis provides the `check_error` command to enable error checking within the compile script. For details about this command, see the man page.

- Exits the shell

  If you do not include the `quit` command to exit the shell, the make process freezes while waiting for the tool to finish.

You must provide custom compile scripts for each chip-level compile run in which you want to override the default.

Example 3-4 shows a custom compile script.

Example 3-4 XG Mode Custom Compile Script

```
# Set the environment variables
source pass0/scripts/env

# Read the design
read_ddc pass0/db/pre_compile/MUX.ddc
if { [check_error -v] == 1} { exit 1 }

# Link the design
current_design MUX
link
if { [check_error -v] == 1} { exit 1 }

# Set the constraints
source pass0/constraints/MUX.con
if { [check_error -v] == 1} { exit 1 }

# Define the compile strategy
set compile_new_boolean_structure false
set_structure true -timing true -boolean false
set_max_area 0
set_flatten false

# Compile the design
compile  -map_effort medium
if { [check_error -v] == 1} { exit 1 }

# Write the design to the destination directory
write -format ddc -output pass0/db/post_compile/MUX.ddc {MUX}
```

Example 3-4 Custom Compile Script (Continued)

```
# Generate reports
report_timing > pass0/reports/MUX.tim
report_area > pass0/reports/MUX.area
report_constraints > pass0/reports/MUX.cstr

# Exit the shell
quit
```

Running the Compile Job

Many aspects of the compile job, such as how the compile job is invoked and which executable file is used, are controlled by the makefile. For information about customizing the makefile, see "Generating the Makefile" on page 3-12.

To further customize how the compile job is run, you can

- Specify the make utility
- Specify the number of parallel compile jobs
- Check out required licenses before running the compile job
- Use GRD to submit the compile jobs
- Run the compile job from the UNIX prompt

The following sections describe these options.

Specifying the Make Utility

By default, Automated Chip Synthesis uses gmake as the make utility. You can use another make utility by changing the value of the `acs_make_exec` variable.

By default, Automated Chip Synthesis uses only the -j option with the gmake command. This option specifies the number of parallel jobs to run. For more information about specifying the number of parallel jobs, see the next section, "Specifying the Number of Parallel Compile Jobs."

Another useful option of the gmake command is -k. By default, if an error occurs in one of the parallel compile jobs, gmake waits for all other currently running jobs to finish, then stops. When you specify the -k option, gmake continues to process all modules that are not dependent on the module that had an error. This means that gmake builds as much of the design as possible while you fix the problem that caused the error. If you complete the fix before gmake is done, the job is automatically restarted (assuming that the fixed file has a more recent time stamp). Otherwise you can rerun gmake to complete processing of the design.

To change the gmake command arguments, set the `acs_make_args` variable.

Specifying the Number of Parallel Compile Jobs

By default, Automated Chip Synthesis runs the compile jobs serially. If your host machine has more than one CPU, you can run the compile jobs in parallel by setting the `acs_num_parallel_jobs` variable to the number of CPUs on your host machine.

Note:

Each compile job uses a tool license. To limit the number of licenses used by Automated Chip Synthesis, limit the number of parallel compile jobs.

If you are using a batch submission command to invoke the compile jobs, the compile jobs are always run in parallel. For more information about controlling how the compile jobs are invoked, see “Generating the Makefile” on page 3-12.

Checking Out Required Licenses

By default, Automated Chip Synthesis does not check out the required licenses (such as Design-Compiler, Test-Compiler or DC-Ultra-Opt) before starting the compile run. If the required licenses are not available at runtime, the compile job fails.

To prevent this from occurring, set the `acs_lic_wait` variable to specify that Automated Chip Synthesis should wait until all required licenses are available before starting a compile run. Set the `acs_lic_wait` variable to the maximum time, in minutes, that Automated Chip Synthesis should wait for licenses. For example, to wait for a maximum of five minutes for required licenses, enter the following command before running a chip-level compile command:

```
dc_shell-xg-t> set acs_lic_wait 5
```

Note:

If you always want to wait for the required licenses, set the `acs_lic_wait` variable in your .synopsys_dc.setup file.

When you set the `acs_lic_wait` variable to a positive value, Automated Chip Synthesis adds the `-wait` and `-checkout` options to the shell invocation commands in the makefile.

Automated Chip Synthesis determines the required licenses from the compile strategy attributes on each partition. For more information about the compile strategy attributes, see “Changing the Compile Strategy” on page 3-29.

If the required licenses are not available after the specified time, an error message is generated and the compile job is not executed.

Running the Compile Job in Batch Mode

When you run the `acs_submit` or `acs_submit_large` command with one or more configuration options (`-host`, `-exec`, or `-command`), you enable batch submission of the compile jobs.

Note:

Running the `acs_submit` or `acs_submit_large` command with only the `-show` option (or with no options) has no effect on how the compile jobs are invoked.

The `acs_submit_large` command defines the compile configuration for the compile partitions specified in the `-partitions` option (if you omit the `-partitions` option, the compile configuration specified by `acs_submit_large` applies to the top-level design). The `acs_submit` command defines the compile configuration for all other compile partitions.

You can use one or both of these commands, depending on whether you require one or two compile configurations. For example, to compile the top-level design on a 64-bit host and the compile partitions on a 32-bit host, enter the following commands:

```
dc_shell-xg-t> acs_submit_large -partitions TOP \
   -host mysparc64 \
   -exec $SYNOPSYS/sparc64/syn/bin/dc_shell
dc_shell-xg-t> acs_submit -host mysparc32 \
   -exec $SYNOPSYS/sparcOS5/syn/bin/dc_shell
```

When you use the `acs_submit` or `acs_submit_large` command to configure the compile jobs, Automated Chip Synthesis generates a csh script that invokes the compile shell for each compile job. The

generated csh scripts are stored in the directory defined for the pass_compile_script file type (default directory is $acs_work_dir/*dest_dir*/scripts).

The following sections describe how to configure the compile jobs when enabling batch submission with these commands.

Specifying the Host Machine. If you do not specify the `-host` option when you run the `acs_submit` or `acs_submit_large` command, Automated Chip Synthesis submits the compile jobs from your host machine.

If you specify the `-host` option, Automated Chip Synthesis submits the compile jobs from the specified host.

Specifying the Batch Submission Command. If you do not specify the `-command` option when you run the `acs_submit` or `acs_submit_large` command, Automated Chip Synthesis uses the following LSF command as the batch submission command:

```
bsub -K -P design_dest_dir -o dest_dir/logs/design.log
```

The `-K` option specifies that the submitted job must finish before returning to the make utility (without this option the make utility cannot successfully compile the design). The `-P` option assigns a project name (default name is *design_dest_dir*) to the submitted job.

The `-o` option specifies the log file that contains the job output. If your batch submission command does not support the `-o` option, you can set the `acs_submit_log_uses_o_option` variable to `false`. This removes the -`o` option from the batch submission command and captures the dc_shell output by using the UNIX tee command.

Note:

LSF is third-party software that is separately licensed. If you use the LSF `bsub` command, the host machine must be licensed to spawn LSF jobs; otherwise the compile job fails. To learn more about the LSF software, see the Platform Computing Web site at http://www.platform.com.

If your configuration supports a batch submission command other than LSF `bsub`, specify the command and its arguments by using the `-command` option.

Important:

If you specify a different batch submission command, you must also specify an option equivalent to the `bsub -K` option to require the submitted job to finish before returning to the make utility. Without this requirement the make utility cannot successfully compile the design.

If you are using Global Resource Director (GRD) rather than LSF, you cannot run the compile jobs simply by changing the batch submission command and its arguments. For details about using GRD with Automated Chip Synthesis, see “Using GRD to Submit Compile Jobs” on page 3-47.

Specifying the Executable File. If you do not specify the `-exec` option when you run the `acs_submit` or `acs_submit_large` command, Automated Chip Synthesis uses the current executable file to compile your design.

If you specify the `-exec` option, Automated Chip Synthesis uses the specified string as the executable file.

Note:

If you are submitting the compile jobs to a heterogeneous compute network, define an ARCH environment variable in your .cshrc file that contains the current architecture, then specify the executable file by using the ARCH variable rather than a hard-coded platform name. For example,

```
dc_shell-xg-t> acs_submit \
   -exec "\$SYNOPSYS/\$ARCH/syn/bin/dc_shell-xg-t"
```

Verifying the Compile Configurations. Both the `acs_submit` and `acs_submit_large` commands have a `-show` option that reports the specified compile configuration. Before running a chip-level compile command, verify the compile configurations by using the `acs_submit -show` and `acs_submit_large -show` commands.

Using GRD to Submit Compile Jobs

Because the GRD `qsub` command does not provide the capability to wait for the submitted job before returning to the make utility (similar to the LSF `bsub -K` option), you must manually synchronize the compile jobs when using GRD. This prevents you from using GRD by simply defining the batch submission command as `qsub`.

The easiest way to use Automated Chip Synthesis with GRD is to create a csh script that waits for a compile job to complete before returning to the make utility and then use this script as the batch submission command. Example 3-5 shows a sample csh script.

Example 3-5 Script to Run ACS with GRD (run_qsub.csh)

```
#! /bin/csh
set compile_scr = $1
set final_ddc = \
   `echo $compile_scr | \
    sed -e "s/scripts/db\/post_compile/g" | \
    sed -e "s/.csh/.ddc/g"`
qsub -cwd -l mem_free=2G $1
while (! -e $final_ddc)
   sleep 10
end
```

Assuming that the csh script is in your project root directory, enter the following command to define the csh script as the batch submission command:

```
dc_shell-xg-t> acs_submit -command ./run_qsub.csh
```

The makefile generated by Automated Chip Synthesis contains a command similar to the following for each compile partition:

```
USER_SUBMIT_CMD = ./run_qsub.csh
 $(USER_SUBMIT_CMD) /user/proj_dir/pass0/scripts/partition.csh
```

Running the Compile Job From the UNIX Prompt

By default, Automated Chip Synthesis runs the compile job from within the synthesis tool. Running the compile job from within the synthesis tool provides a single environment for the chip-level compile run; however, this method requires a Design-Compiler license (depending on which flow you use) for the duration of the compile job, plus an additional license for each active compile job. In addition, monitoring the status of the compile jobs is difficult. You can use the chip-level compile commands to generate the script files, then invoke the make utility directly from the UNIX prompt.

If you do not want the compile job to run from within the synthesis tool, use the `-prepare_only` option when invoking the chip-level compile command (`acs_compile_design`, `acs_refine_design`, or `acs_recompile_design`).

Use a make utility, such as gmake, to run the makefile from the UNIX command line. The gmake utility can run either serial or parallel compile jobs. By default, gmake runs serial compile jobs.

For example, to run serial compile jobs using $acs_work_dir/pass0 as the destination directory, enter the following commands:

```
dc_shell-xg-t> acs_compile_design -prepare_only design

% gmake -f pass0/Makefile >& pass0/logs/gmake.log
```

To run parallel compile jobs, specify the `-j` *n* option, where *n* is the number of CPUs available on the host machine.

Caution!

If you are performing parallel compiles on multiple machines, a problem can occur if the system clocks on these machines are not synchronized. The gmake utility issues a message that "modification is in the future". If you are running gmake version 3.76 or greater, you can ignore this message (the compile job continues). If you are running an earlier version of gmake, the compile job might terminate and you will have to rerun the compile job.

For example, to run parallel compile jobs using $acs_work_dir/pass1 as the destination directory, enter the following commands:

```
dc_shell-xg-t> acs_refine_design -prepare_only design

% gmake -f pass1/Makefile -j n >& pass1/logs/gmake.log
```

If you are using LSF, you might want to use some of the common LSF commands listed in Table 3-6 to manage the parallel compile runs. For complete information about the LSF command set, see the documentation available at the Platform Computing Web site at http://www.platform.com.

Table 3-6 Common LSF Commands

Command	Description
bjobs -P *project_id* -u all -l	Monitors jobs in the queue. By default, all jobs are displayed. Displays the jobs in the specified project. Displays the jobs for all users. Displays a detailed description for each job.
bpeek *job_id*	Enables viewing of the specified job output before the job completes.
bkill *job_id*	Terminates the specified job.

Note:

The `bjobs` command displays the *job_id* values.

Modifying the Automated Chip Synthesis Flow

You can modify the Automated Chip Synthesis flow by

- Adding chip-level compile runs
- Updating your design
- Changing the chip-level compile commands

The following sections describe these methods.

Adding Chip-Level Compile Runs

You can modify the default flow by performing additional chip-level compile runs.

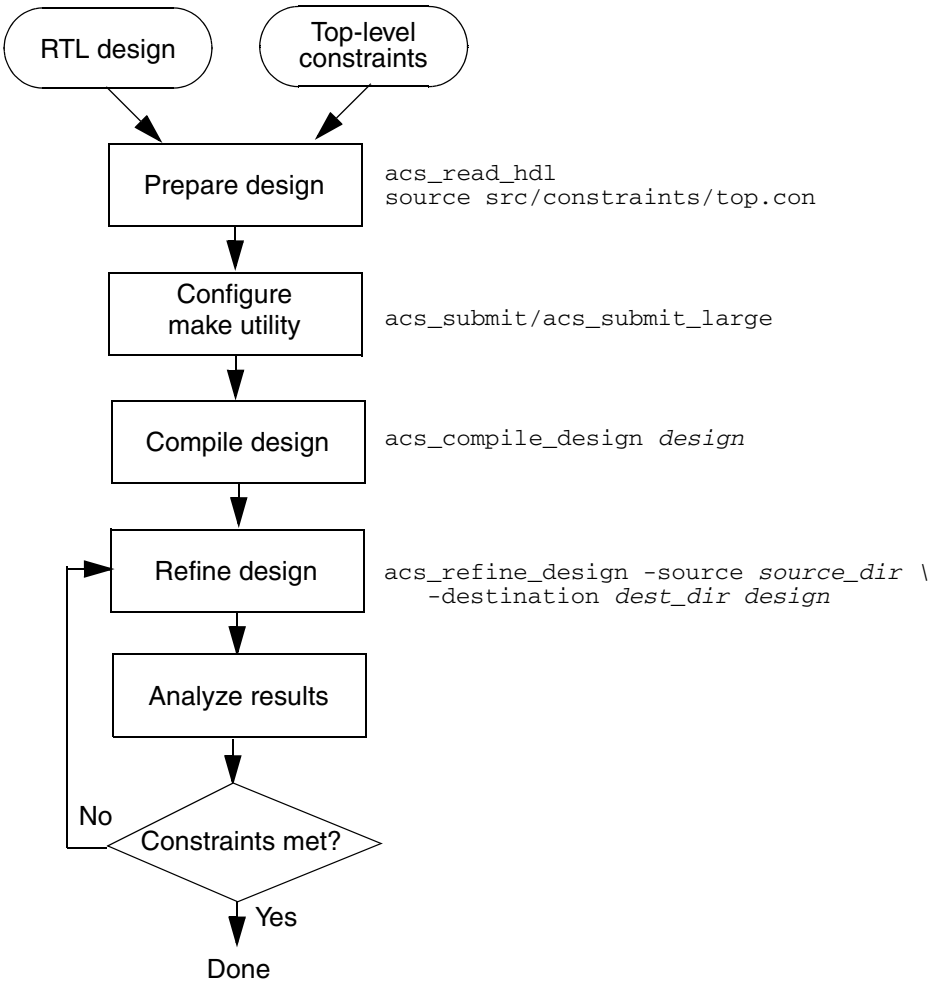
- If the compiled design is close to meeting its constraints, you can usually meet constraints by continuing to refine the design.
- If your design has very tight timing constraints, you might want to improve the starting point by recompiling the GTECH design database using design budgets instead of propagated constraints.

The following sections describe these options.

Refining Your Design

If your design does not meet constraints after you run the `acs_refine_design` command, you can rerun the command as many times as necessary to meet constraints (as shown in Figure 3-2). For details about refining your design, see “Refining Your Design” on page 2-35.

Figure 3-2 Adding Refine Runs



Recompiling Your Design

In most cases the compiled design that you generated using RTL budgeting or propagated constraints provides a good starting point for refinement. In some cases you can get better results if you recompile the GTECH design database using design budgets generated from the compiled design.

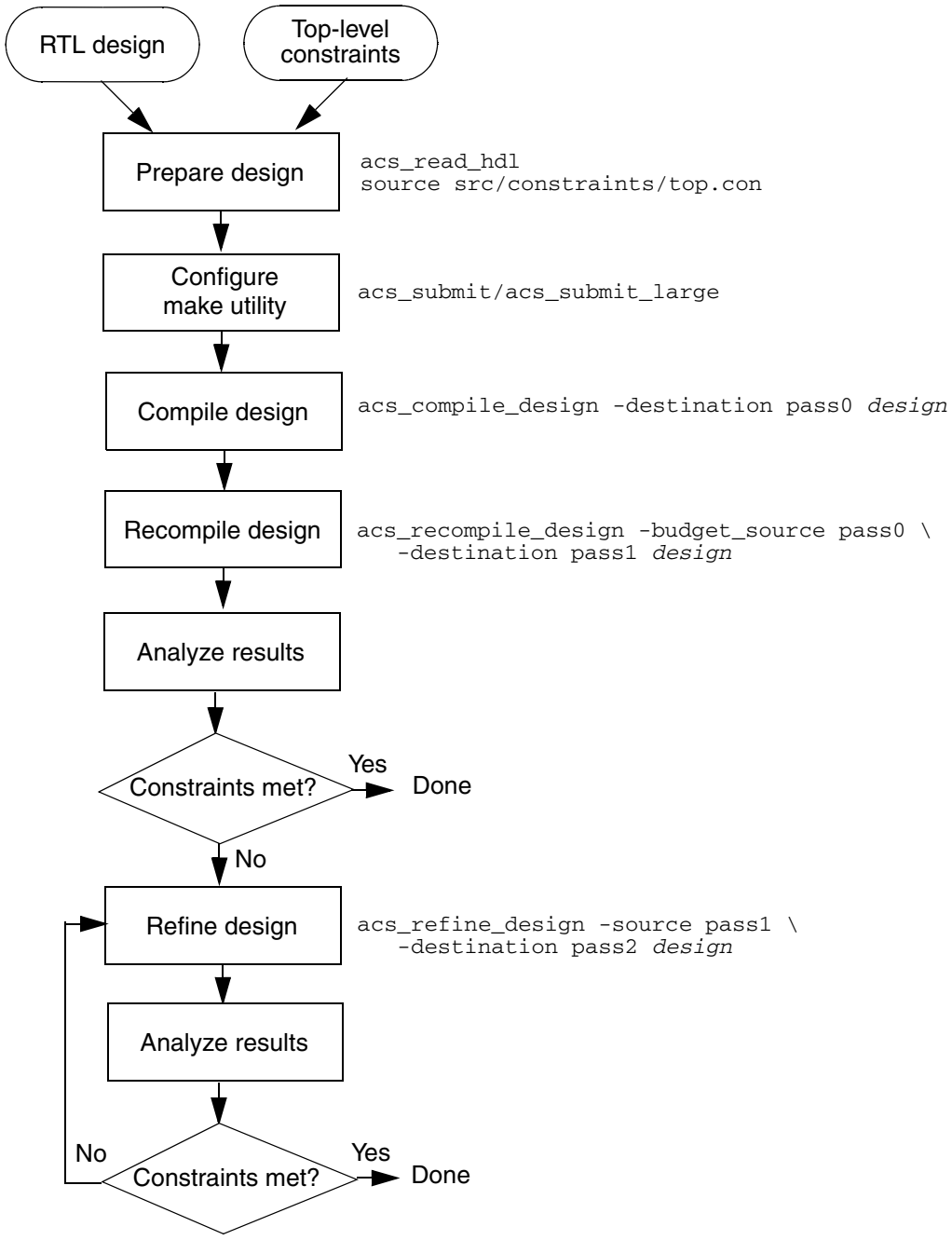
Use the `acs_recompile_design` command to recompile the GTECH design database to your target technology library.

```
dc_shell-xg-t> acs_recompile_design \
   -budget_source budget_dir -destination dest_dir design
```

The `acs_recompile_design` command uses the results of a previous compile run to generate the design budget, then uses this design budget when compiling the design. You must specify both the budgeting source directory (`-budget_source` *`budget_dir`* option) and the destination directory (`-destination` *`dest_dir`* option) when invoking the `acs_recompile_design` command.

Figure 3-3 shows the Automated Chip Synthesis design flow when you recompile your design.

Figure 3-3 Adding a Recompile Run

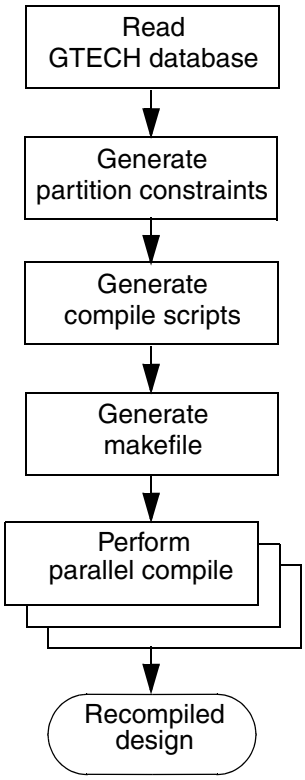


The `acs_recompile_design` command performs the tasks shown in Figure 3-4.

Note:

If a makefile exists in the destination directory ($acs_work_dir/*dest_dir*/Makefile), the `acs_recompile_design` command performs only the parallel compile task (unless you specify the `-force` option).

Figure 3-4 Recompile Design Flow



The following sections describe the default behavior of these tasks. You can customize the `acs_recompile_design` tasks in the same ways that you can customize the tasks in the other chip-level compile commands.

Reading the GTECH Database. The `acs_recompile_design` command first checks for the specified design in memory. If it does not locate the design in memory, it reads the precompiled design database (pre_ddc file type, or pre_db file type if you are running in DB mode) from the $acs_work_dir/pass0 directory tree. This database already has the multiple instances resolved and the compile partitions defined.

If your GTECH database is in a different location, use the `-source` *`source_dir`* option to specify the source directory.

Generating Partition Constraints. The `acs_recompile_design` command generates the partition constraints by performing design budgeting on the design located in the $acs_work_dir/*budget_dir*/db/post_compile directory (where *budget_dir* is the name of the budgeting source directory). For information about design budgeting, see "Generating Partition Constraints for Gate-Level Designs" on page 2-37.

Generating the Compile Scripts. By default, the `acs_recompile_design` command creates compile scripts for the top-level design and for each compile partition. For information about compile partitions, see "Identifying Compile Partitions" on page 2-18.

Automated Chip Synthesis places the generated script files in the $acs_work_dir/*dest_dir*/scripts directory (where *dest_dir* is the name of the destination directory).

The default partition compile script (shown in Example 3-6) performs a medium-effort compile followed by a high-effort incremental compile on each compile partition. The default top-level compile script (shown in Example 3-7) performs a medium effort compile on the top-level design, followed by a top-level compile. Before

performing a full compile, Automated Chip Synthesis places a `dont_touch` attribute on each subpartition of the design being compiled. Automated Chip Synthesis removes the `dont_touch` attributes (except on the reference designs for multiple instances) after completing the full compile (before starting the boundary compile).

Example 3-6 Default Partition Compile Script (acs_recompile_design)

```
# full compile
set_max_area 0.0
set_flatten false
compile -map_effort medium
```

Example 3-7 Default Top-Level Compile Script (acs_recompile_design)

```
# full compile
set_max_area 0.0
set_flatten false
compile -map_effort medium

# boundary compile
set compile_top_acs_partition true
set compile_top_all_paths true
compile -top
```

Generating the Makefile. A makefile is a special script that defines the commands and dependencies required to build a target (in this case, compile a design). You automatically build your target by running a make utility. The make utility can be one of the make utilities supplied with UNIX or one supplied by a third-party program. By default, Automated Chip Synthesis uses gmake as the make utility.

Automated Chip Synthesis automatically generates a makefile to compile your design and places it in the $acs_work_dir/*dest_dir* directory (where *dest_dir* is the name of the destination directory). The generated makefile is designed to run on the UNIX gmake utility.

Running the Compile Job. If your configuration supports it, Automated Chip Synthesis runs the partition compile jobs in parallel. The compile process requires a Design-Compiler license for the duration of the compile job, plus a Design-Compiler license for each active compile job.

The `acs_recompile_design` command writes the compiled designs (top-level design and the compile partitions) to the $acs_work_dir/*dest_dir*/db/post_compile directory (where *dest_dir* is the name of the destination directory).

Updating Your Design

If you make a small change to part of your design, you can use Automated Chip Synthesis to incrementally update your design. Automated Chip Synthesis recompiles only the affected compile partitions and does not recompile the entire design.

The basic steps required to update to your design are

1. Modify the HDL source code.
2. Update the modified subdesigns in the unmapped design.
3. Merge the modified subdesigns with the mapped design.
4. Recompile the changed portions of the design.

There are several ways to implement this flow. The spectrum goes from a simplified update flow that saves only the design database associated with the latest update iteration, to a more complex flow that saves the design databases associated with each update iteration.

The following sections describes the basic steps in the update flow and how to implement these two update flows. You can use these example flows to build an update flow that meets your needs.

Modifying the HDL Source Code

Modify the HDL source code for any subdesigns that have changed.

Important:

If you used the `uniquify` command to resolve multiple instances in your design, the modified subdesign names must match the unique (rather than the original) subdesign names. In addition, the instantiations of the uniquified designs in their parent design must use the unique subdesign names.

Updating the Unmapped Design

After updating the HDL source code for the modified subdesigns, use the `acs_read_hdl` command in update mode to input the modified HDL source code and update the unmapped design.

The `acs_read_hdl` command provides two update modes:

- Automatic

  In automatic update mode, the `acs_read_hdl` command automatically determines the modified source files by comparing the time stamps of the files in the HDL source directory with the time stamp of the unmapped design file (*top_design*.ddc) in the

elab_ddc directory (default path is $acs_work_dir/elab/db). If you are running in DB mode, the `acs_read_hdl` command checks against the *top_design*.db file in the elab_db directory.

To invoke `acs_read_hdl` in automatic update mode, use the `-auto_update` option.

Note:

Automatic update mode requires files generated by the update mode of `acs_read_hdl`. If you have not previously run `acs_read_hdl` in update mode (either automatic or manual), `acs_read_hdl -auto_update` inputs the entire design, instead of just the modified subdesigns.

- Manual

  In manual update mode, you must explicitly specify the modified source files. The `acs_read_hdl` command inputs only the specified files.

  To invoke `acs_read_hdl` in manual update mode, use the `-update` *`modified_file_list`* option, where *modified_file_list* specifies the HDL source files to be input.

Important:

Automatic update mode does not support uniquified or ungrouped designs. If you used the `uniquify` or `ungroup` command to resolve multiple instances in your design, you must use the manual update mode to specify these designs.

You can use the update modes independently or simultaneously. For example, if some, but not all, of the updated designs have been uniquified, use the following command to update the unmapped design:

```
dc_shell-xg-t> acs_read_hdl -auto_update \
   -update uniquified_designs
```

For more information about the `acs_read_hdl` command, see “Inputting the RTL Design” on page 2-4.

Merging the Modified Subdesigns

Use the `acs_merge_design` command to merge the modified subdesigns with the mapped design. To merge the modified subdesigns, the `acs_merge_design` command performs the following tasks:

1. Removes all designs from memory
2. Loads the mapped design
3. Loads the unmapped design
4. For each changed subdesign and its parent compile partitions, replaces the mapped design with its GTECH implementation
5. Saves the resulting design

Loading the Mapped Design. The mapped design is the result of a previous chip-level compile command or update iteration.

By default, `acs_merge_design` loads the mapped design from the postcompile directory (the post_ddc directory in XG mode or the post_db directory in DB mode) of the pass0 data directory (default path is $acs_work_dir/pass0/db/post_compile).

If the mapped design is not located in the pass0 data directory, use the `-mapped` *`mapped_dir`* option to specify the data directory that contains the mapped design. If the mapped design is located in the precompile directory (the pre_ddc directory in XG mode or the pre_db directory in DB mode), rather than the postcompile directory (the post_ddc directory in XG mode or the post_db directory in DB mode), specify `-type pre`.

Loading the Unmapped Design. The unmapped design is the updated unmapped design you just created by using the `acs_read_hdl` command.

By default, `acs_merge_design` loads the unmapped design from the elaborated design directory (the elab_ddc directory in XG mode or the elab_db directory in DB mode). The default path for the elaborated design directory is $acs_work_dir/elab/db. This location is also the default destination directory for the `acs_read_hdl` command. If you used the default destination directory when you ran `acs_read_hdl`, use the default unmapped directory. If you did not use the default destination directory when you ran `acs_read_hdl`, you must use the `-unmapped` *`unmapped_dir`* option to specify the directory that contains the unmapped design.

Merging the Design. In the merge process, `acs_merge_design` uses the unmapped version of each modified subdesign and its parent compile partitions. For all other subdesigns, the `acs_merge_design` command uses the mapped version.

You can either let Automated Chip Synthesis identify the modified subdesigns (`-auto_update` option), or you can explicitly specify the modified subdesigns (`-update` *`subdesign_list`* option).

When you specify the `-auto_update` option, `acs_merge_design` finds the modified subdesigns by comparing the time stamps of the unmapped designs with the earliest time stamp of the files in the pass0 precompile directory (the pre_ddc directory in XG mode or the pre_db directory in DB mode). The default precompile directory path is $acs_work_dir/pass0/pre_compile. If you used a data directory other than pass0 for the most recent `acs_compile_design` or `acs_recompile_design` command, use the `-reference` *`ref_dir`* option to specify the data directory. When you specify the `-reference` option, `acs_merge_design` uses the precompile directory of the *ref_dir* data directory for the time stamp comparison.

Saving the Merged Design. The merged design remains in memory and is stored in an Automated Chip Synthesis data directory.

By default, `acs_merge_design` writes the merged design to the precompile directory of the pass0_incr data directory (default path is $acs_work_dir/pass0_incr/db/pre_compile).

To save the merged design in the precompile directory of a different data directory, use the `-destination` *`dest_dir`* option to specify the destination data directory.

Recompiling the Changes

Use the `acs_compile_design -update` command to recompile the design. The `acs_compile_design -update` command performs the following tasks:

1. Reads the merged design

   The `acs_compile_design -update` command first checks for the specified design in memory. If it does not locate the design in memory, it reads the design from the pass0_incr precompile directory (default path is $acs_work_dir/pass0_incr/db/pre_compile).

   If the merged design is not in one of these locations, use the `-update_source` option to specify the data directory containing the merged design. Automated Chip Synthesis looks for the merged design in the precompile directory of the specified data directory.

   If Automated Chip Synthesis does not find the merged design, it generates an error message and exits.

2. Generates the partition constraints

   The `acs_compile_design -update` command generates constraints only for the parent compile partitions of the specified subdesigns. Automated Chip Synthesis does not regenerate constraints for compile partitions that are not parents of the specified subdesigns. These compile partitions are considered to be fixed during budgeting.

3. Generates the compile scripts

   The `acs_compile_design -update` command generates compile scripts only for the parent compile partitions of the specified subdesigns and the top-level design.

4. Generates the makefile

   The makefile generated by the `acs_compile_design -update` command compiles the parent compile partitions of the specified subdesigns and performs a boundary compile (`compile -top`) on the top-level design. Although the other compile partitions might be modified during the top-level boundary compile, they are not individually recompiled.

5. Performs a parallel compile

6. Saves the updated design

   The updated design database consists of the recompiled compile partitions and the top-level design, plus copies of the unchanged compile partitions. The unchanged compile partitions are copied from the destination precompile directory (*dest_dir*/db/pre_compile).

   By default, *dest_dir* is the pass0_incr directory. To specify a different destination data directory, use the `-destination` *`dest_dir`* option.

   The updated design is saved in the postcompile directory of the destination data directory (default path is $acs_work_dir/pass0_incr/db/post_compile).

Simplified Update Flow

This method uses the same data directory to store the design databases associated with each iteration (original `acs_compile_design` results or an update iteration).

Using the same data directory simplifies the flow, because you can use the same set of commands to run each update iteration. However, using the same data directory for each update iteration means that you overwrite the data from the previous iteration and lose control of the update history. To save the results of an update iteration, you must manually copy the design database files (.ddc files in XG mode or .db files in DB mode) to a backup location.

When using the simplified update flow, you are using an existing data directory that contains the makefile from a previous run. You must specify the `-force` option with the `acs_compile_design` command to generate new partition constraints, new compile scripts, and a new makefile before compiling the updated design. Otherwise, Automated Chip Synthesis uses the existing makefile to perform the parallel compile without updating the partition constraints or compile scripts.

The commands used to perform an update iteration for pass0 using the simplified flow are

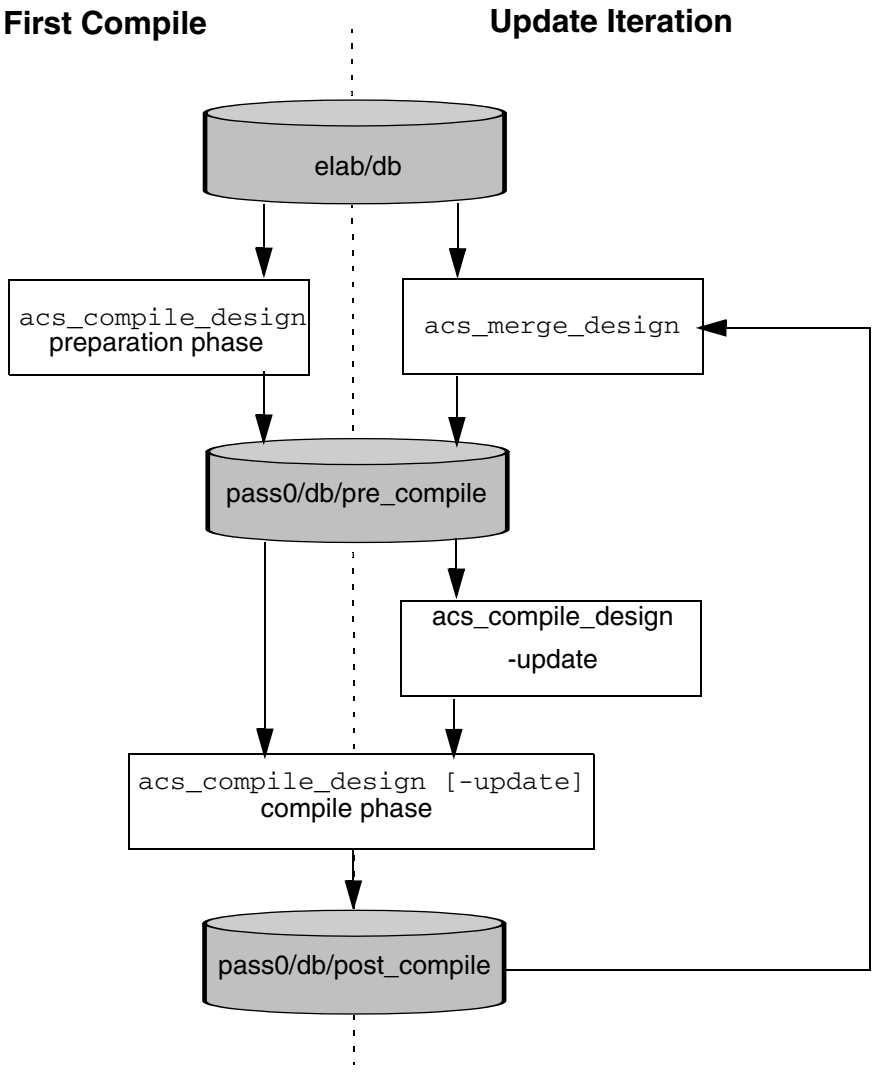
```
acs_read_hdl -auto_update -hdl_source source_files design
acs_merge_design design -auto_update
acs_compile_design -update -force design
```

You would use similar commands to perform an update iteration for a subsequent pass. In the general case, the source directory for the modified subdesigns (as specified by the `-unmapped` option) can be either the elaborated design directory (default location is

$acs_work_dir/elab/db), the precompile directory (default location is $acs_work_dir/pass{*n*-1}/db/pre_compile), or the postcompile directory (default location is pass{*n*-1}/db/post_compile).

Figure 3-5 shows the data flow for the simplified pass0 update flow.

Figure 3-5 Simplified Update Flow



Update Flow That Saves All Design Changes

This method uses a new data directory for each iteration pass. Although this method requires more disk space, it keeps a complete history of the design changes and allows you to revert to any previous state of the design. To save space, you can use the `remove_pass_directories` command to remove unneeded data directories.

When you use this update flow, you must specify the input files for each step in the update process. Each iteration uses different locations for the input files.

Important:

After updating your design, you must specify the update destination directory as the source directory for subsequent passes. You cannot use the default source directory.

The commands used to perform the first update iteration for pass0 using this flow are

```
acs_read_hdl -auto_update design
acs_merge_design design -auto_update \
   -mapped pass0 -unmapped elab/db \
   -destination pass0_incr1
acs_compile_design design -update \
   -update_source pass0_incr1 \   -destination pass0_incr1
```

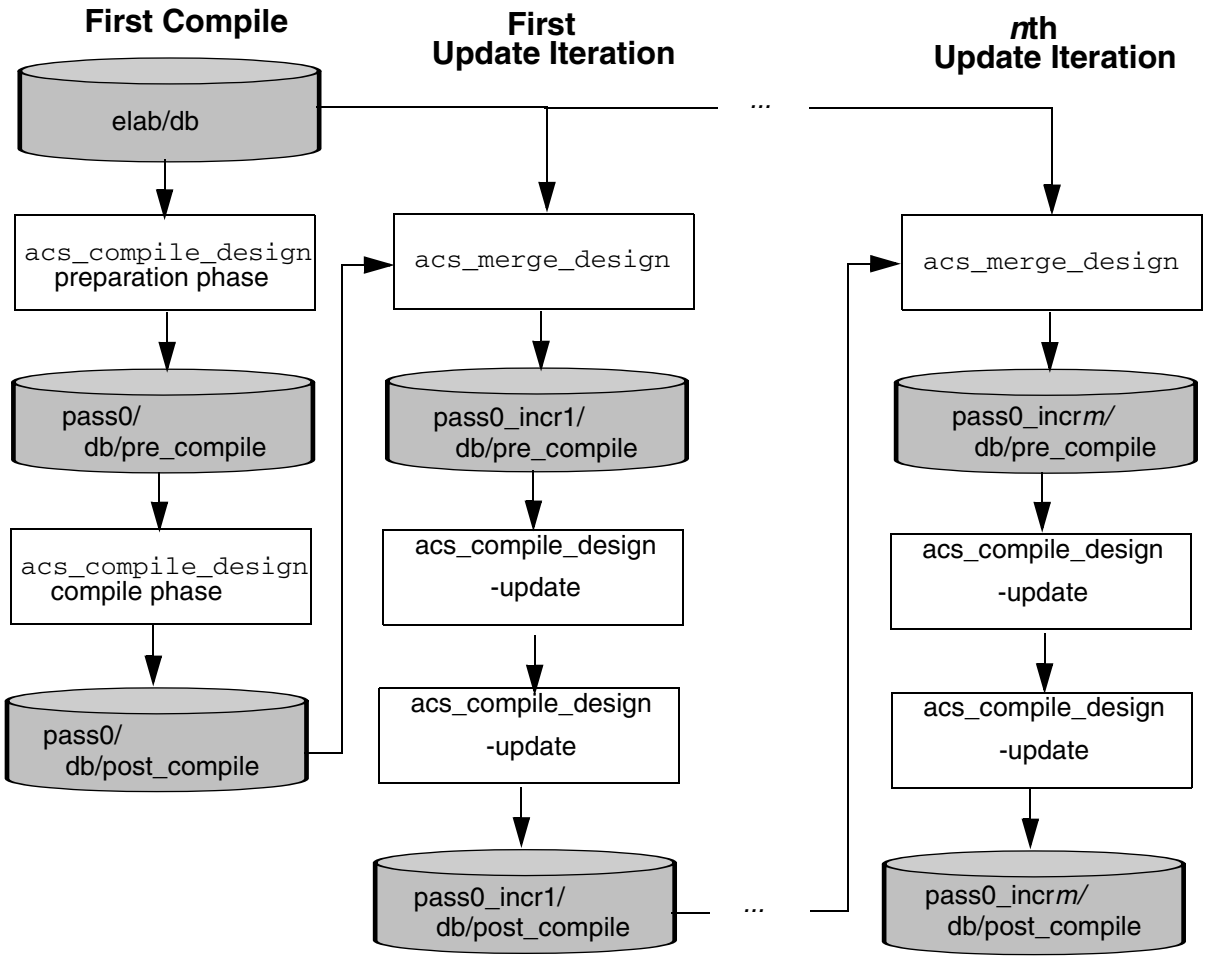
The commands used to perform additional update iterations for pass0 using this flow are

```
acs_read_hdl -auto_update design
acs_merge_design design -auto_update \
   -mapped pass0_incr{m-1} -unmapped elab/db \
   -destination pass0_incr{m}
acs_compile_design design -update \
   -update_source pass0_incr{m} \
   -destination pass0_incr{m}
```

You would use similar commands to perform an update iteration for a subsequent pass. In the general case, the source directory for the modified subdesigns (as specified by the `-unmapped` option) can be either the elaborated design directory (default location is $acs_work_dir/elab/db), the precompile directory (default location is $acs_work_dir/pass{*n*-1}/db/pre_compile), or the postcompile directory (default location is pass{*n*-1}/db/post_compile).

Figure 3-6 shows the pass0 data flow for this update flow.

Figure 3-6 Update Flow That Saves All Design Data



Changing the Chip-Level Compile Commands

You can override the default behavior of the chip-level compile commands (`acs_compile_design`, `acs_recompile_design`, and `acs_refine_design`) by providing custom Tcl procedures. This section describes the internal behavior of the chip-level compile commands and how to customize these commands.

About the Chip-Level Compile Commands

Earlier in this document, each of the chip-level compile commands was described in terms of the tasks they performed.

Table 3-7 shows the differences between the three chip-level compile commands.

Table 3-7 Comparison of Chip-Level Compile Commands

Command	Budgeting source directory	Compile source directory	Compile mode
`acs_compile_design`	elab/db (RTL)	elab/db	default
`acs_refine_design`	pass*n*/db/post_compile	pass*n*/db/post_compile	incremental
`acs_recompile_design`	pass*n*/db/post_compile	pass*n*/db/pre_compile	default

Each command has a similar flow, but the implementation details vary. Figure 3-7 shows the basic design flow, including tasks and data flow. The destination directory is represented as *dest_dir* in this figure. The destination directory is a subdirectory of the project root directory ($acs_work_dir).

Table 3-8 shows the Automated Chip Synthesis commands used to implement each of the tasks shown in Figure 3-7. See the man pages for details about these commands.

Figure 3-7 Chip-Level Compile Flow

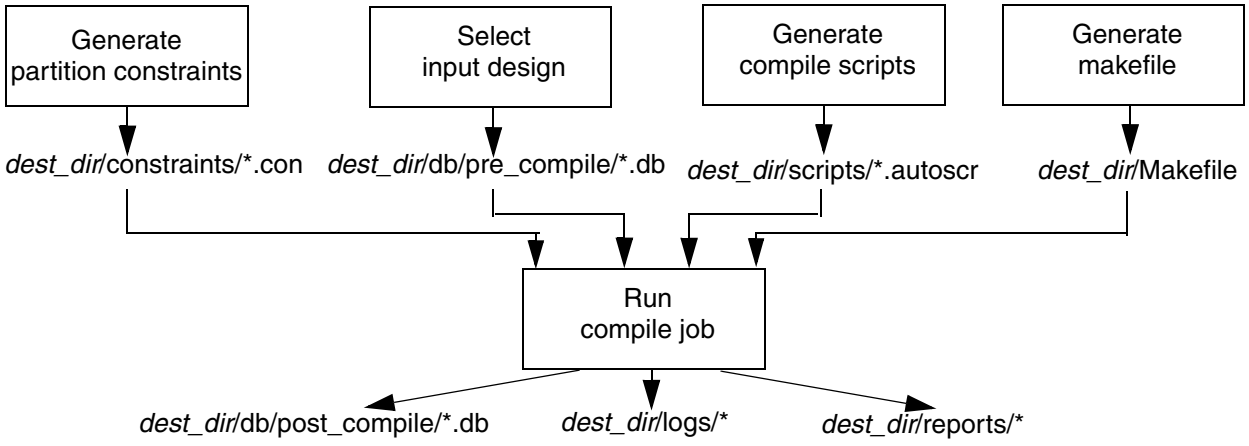


Table 3-8 Chip-Level Compile Tasks

Task	Command	Output files
Generating partition constraints	`dc_allocate_budgets` `write_partition_constraints`	*dest_dir*/constraints/*.con
Selecting the input design	`read_ddc` or `read_db` or `read_partition`, `write_partition`	*dest_dir*/db/pre_compile/*.ddc or *dest_dir*/db/pre_compile/*.db
Generating the compile script	`write_compile_script`	*dest_dir*/scripts/*.autoscr
Generating the makefile	`write_makefile`	*dest_dir*/Makefile
Running the compile job	`compile_partitions`	*dest_dir*/db/post_compile/*.ddc or *dest_dir*/db/post_compile/*.db *dest_dir*/logs/*.log *dest_dir*/reports/*.tim *dest_dir*/reports/*.area *dest_dir*/reports/*.cstr

The following examples (Example 3-8, Example 3-9, and Example 3-10) show the commands used to implement each chip-level compile command. The examples do not provide the complete text of the Tcl source files, just the command flow. For the complete text, see the $SYNOPSYS/auxx/syn/acs/*cmd*.tcl files. For details about these commands, see the man pages.

Example 3-8 XG Mode Commands Composing acs_compile_design

```
# Select input design
read_ddc
current_design
link
set_compile_partitions
write_partition

# Generate partition constraints
dc_allocate_budgets -mode rtl
write_partition_constraints

# Generate compile scripts
# (set compile attributes)
write_compile_script

# Generate makefile
write_makefile

# Run compile job
compile_partitions
```

Example 3-9 Commands Composing acs_recompile_design

```
# Generate partition constraints
dc_allocate_budgets -mode gate
write_partition_constraints

# Select input design
remove_design -all
read_partition -type pre
current_design
link
write_partition

# Generate compile scripts
# (set compile attributes)
write_compile_script

# Generate makefile
write_makefile

# Run compile job
compile_partitions
```

Example 3-10 Commands Composing acs_refine_design

```
# Generate partition constraints
dc_allocate_budgets -mode gate
write_partition_constraints

# Select input design
remove_design -all
read_partition -type post
current_design
link
write_partition

# Generate compile scripts
# (set compile attributes)
write_compile_script

# Generate makefile
write_makefile

# Run compile job
compile_partitions
```

Customizing a Chip-Level Compile Command

You must customize the chip-level compile commands before you invoke dc_shell; otherwise Automated Chip Synthesis uses the default, not the customized, implementations.

To generate a custom procedure for a chip-level compile command,

1. Copy the template script from $SYNOPSYS/auxx/syn/acs/*cmd*.template to $acs_work_dir/scripts/acs, and change its suffix from .template to .tcl.

   ```
   % cp $SYNOPSYS/auxx/syn/acs/cmd.template \
      $acs_work_dir/scripts/acs/cmd.tcl
   ```

Note:

If you customized the directory structure, replace $acs_work_dir/scripts/acs with the directory location for the user_compile_script file type.

2. Modify the Tcl source as necessary.

Caution!

Do not change the command line arguments (as defined in the `define_proc_attributes` statement). Doing so prevents the command from working properly.

If you are using customized chip-level compile commands, Automated Chip Synthesis generates an information message each time you invoke a modified command.

Cleaning Up the Data Directories

Automated Chip Synthesis provides two methods of cleaning up the data directories:

- Removing selected files
- Removing an entire data directory

The following sections describe these options.

Removing Selected Files

If you want to use an existing data directory when you rerun a chip-level compile command, you can remove just the following files:

- The compile scripts ($acs_work_dir/*data_dir*/scripts/*.autoscr)
- The constraint files ($acs_work_dir/*data_dir*/constraints/*.con)
- The postcompile design database files ($acs_work_dir/*data*_dir/db/post_compile/*.ddc in XG mde or $acs_work_dir/*data*_dir/db/post_compile/*.db in DB mde)

You do this by using the clean target in the previously generated makefile. The clean target also removes the generated log files ($acs_work_dir/*data_dir*/logs/*) and the generated report files ($acs_work_dir/*data_dir*/reports/*).

```
% gmake -f data_dir/Makefile clean
```

Note:

You must run this command from the UNIX command line. You cannot perform a partial cleanup of a data directory within dc_shell.

Removing a Data Directory

To remove a data directory and all its contents, use the `remove_pass_directories` command.

```
dc_shell-xg-t> remove_pass_directories data_dir
```

4

Automated Chip Synthesis Tutorial

This chapter provides a tutorial that shows how to use the basic steps of Automated Chip Synthesis to optimize your design. Many times, these basic steps will complete your design process. In other cases, you might have to use the advanced capabilities of Automated Chip Synthesis to meet your design goals. For information about these advanced capabilities, see Chapter 3, “Customizing Automated Chip Synthesis.”

The tutorial is presented in the following sections:

Preparing to Run the Tutorial

The following sections describe the tasks you must complete before you can run the Automated Chip Synthesis tutorial:

- Meeting the Prerequisites
- Creating the Tutorial Directories
- Browsing the Setup File

Meeting the Prerequisites

Before you run the Automated Chip Synthesis tutorial, be sure the following conditions are met:

- You have the DC-Expert or DC-Ultra product.
- The .synopsys_dc.setup file in your home directory is compatible with dctcl mode.

  Automated Chip Synthesis runs only in dctcl mode. If your .synopsys_dc.setup file is not compatible with dctcl mode, use the dc-transcript utility to convert it.

Creating the Tutorial Directories

To create the tutorial directories,

1. Change to your home directory.

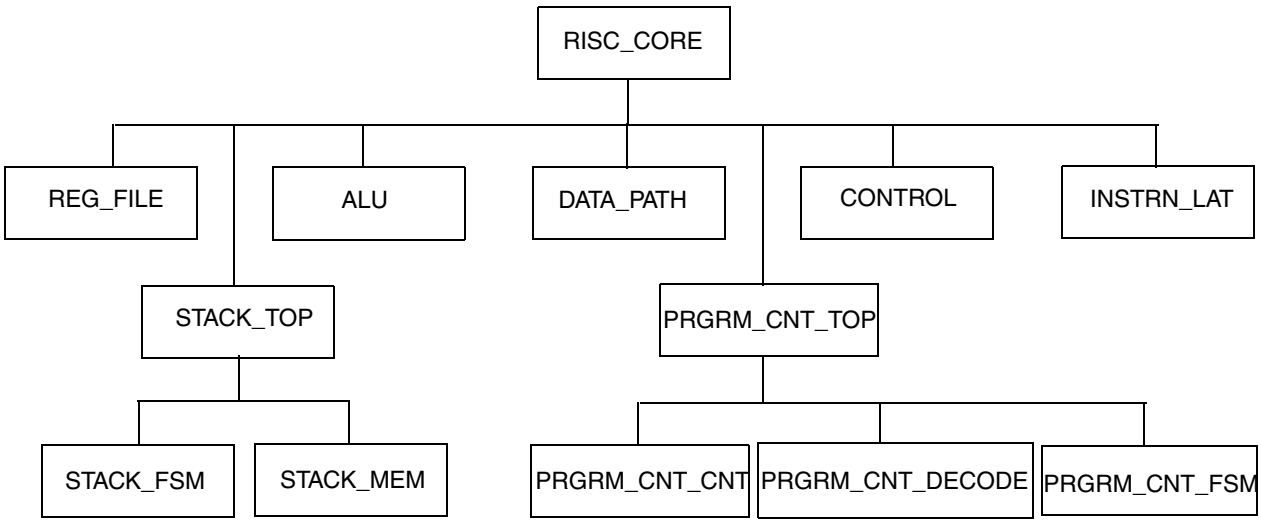
   ```
   % cd
   ```

2. Copy the files from the tutorial directory.

   ```
   % cp -r $SYNOPSYS/doc/syn/acs_tutorial .
   ```

This tutorial uses a simple processor design called RISC_CORE. The design compiles quickly and demonstrates the principles of using Automated Chip Synthesis. Figure 4-1 shows the hierarchy of the RISC_CORE design.

Figure 4-1 Tutorial Design Hierarchy



Note:

The STACK_MEM subdesign is instantiated three times in the STACK_TOP design.

Browsing the Setup File

Example 4-1 shows the .synopsys_dc.setup file provided in the tutorial directory. This setup file imports the `SYNOPSYS` environment variable, defines the synthesis libraries, and specifies the executable location for dc_shell. Use the following commands to browse this file:

```
% cd acs_tutorial
% cat .synopsys_dc.setup
```

If necessary, modify the setup file to reflect your environment before starting the tutorial.

Example 4-1 .synopsys_dc.setup File

```
# .synopsys_dc.setup file for ACS tutorial

set SYNOPSYS [getenv {SYNOPSYS}]

# Make sure that . and ./scripts are in your search path.
# Make sure that * is in your link_library.
# Edit the target_library and link_library appropriately for your design.
#
# NOTE: The tc6a_cbacore library is in $SYNOPSYS/libraries/syn, which is
# in the search path by default. If you do not use the default search path,
# you will have to add the above directory manually.
#
set target_library "tc6a_cbacore.db"
set link_library "* tc6a_cbacore.db dw01.sldb dw02.sldb dw03.sldb"
set synthetic_library "standard.sldb dw01.sldb dw02.sldb dw03.sldb"
set search_path "$search_path scripts"

# The executables can be defined here.
# These are overrides and will default to $SYNOPSYS/sparcOS5/syn/bin
# if not specified.
#
# dc_shell
#set acs_dc_exec PUT_PATH_HERE
```

Running the Tutorial

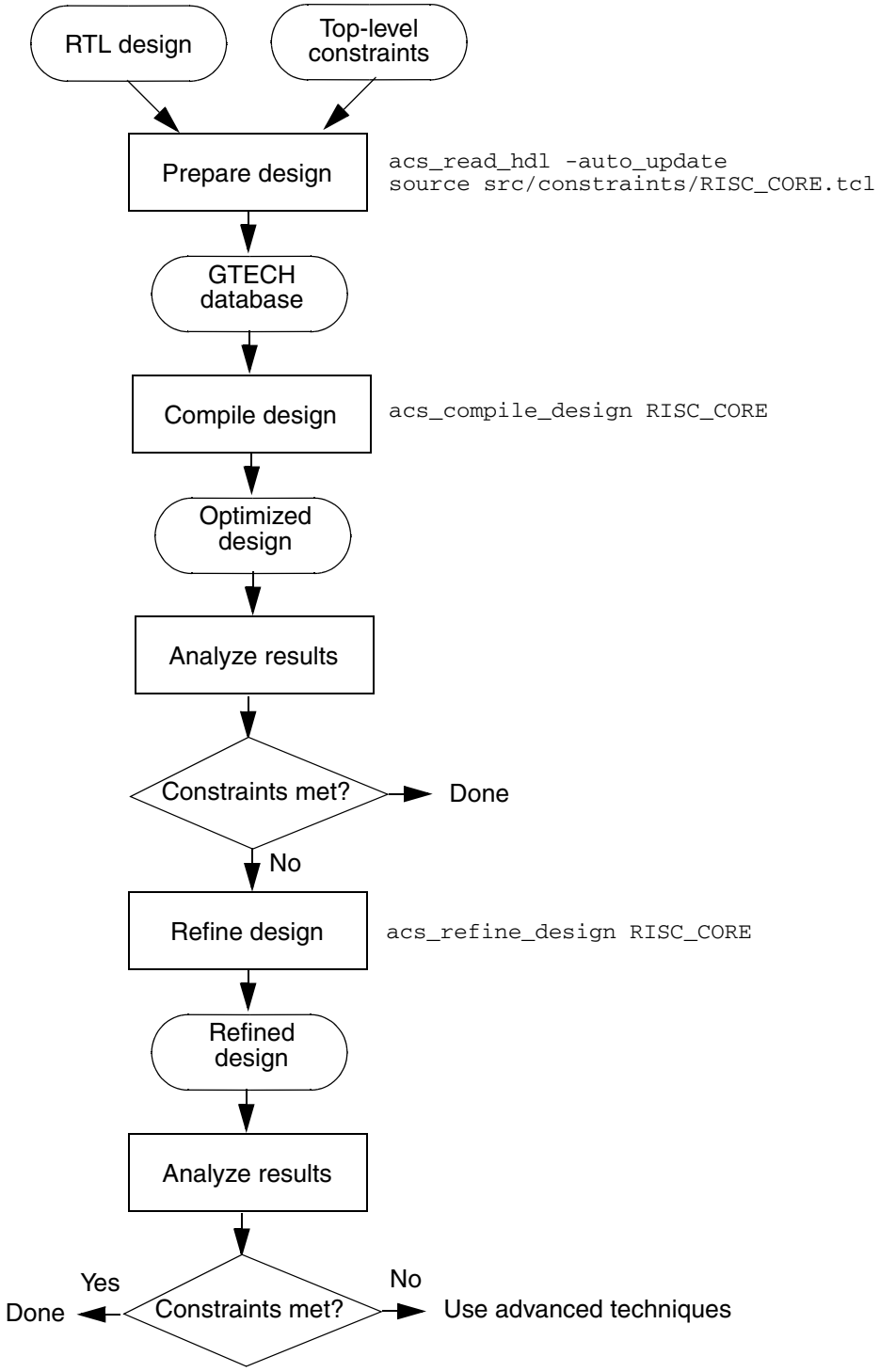
This tutorial follows the default flow shown in Figure 4-2 and uses the Design Compiler tool. The flow is described in the following sections:

1. Preparing the Design
2. Compiling the Design
3. Analyzing the Compiled Design
4. Refining the Design
5. Analyzing the Refined Design

Note:

This tutorial covers only the default Automated Chip Synthesis flow. For information about the advanced techniques, see Chapter 3, “Customizing Automated Chip Synthesis.”

Figure 4-2 Automated Chip Synthesis Design Flow



Preparing the Design

To prepare the design,

1. Create a work directory to store the intermediate files.

   ```
   dc_shell-xg-t> file mkdir work
   dc_shell-xg-t> define_design_lib DEFAULT -path work
   ```

   The Tcl built-in file command creates the work subdirectory, and then the `define_design_lib` command maps the DEFAULT design library to this subdirectory. When the RTL files are analyzed, the intermediate files are placed in the DEFAULT design library.

2. Add the directory containing the Verilog include file to the `search_path` variable. (Verilog only)

   ```
   dc_shell-xg-t> lappend search_path "src/rtl/verilog"
   ```

   HDL Compiler requires that all Verilog include files are located in the search path. If the include file location is not included in the search path, HDL Compiler cannot analyze the design.

3. Input the HDL design.

   ```
   dc_shell-xg-t> acs_read_hdl -auto_update -format fmt \
      -hdl_source src/rtl/fmt RISC_CORE
   ```

   Note:

   To run the Verilog tutorial, replace *fmt* with verilog. To run the VHDL tutorial, replace *fmt* with vhdl.

   The `acs_read_hdl -auto_update` command reads all of the HDL files (*.v when *fmt* is verilog or *.vhd when *fmt* is vhdl) located in the src/rtl/*fmt* directory, then links the top-level design (RISC_CORE). The `acs_read_hdl -auto_update`

command saves the unconstrained GTECH design database in the elab_ddc directory ($acs_work_dir/elab/db). For more information about the `acs_read_hdl` command, see “Inputting the RTL Design” on page 2-4.

4. Apply the top-level constraints.

```
dc_shell-xg-t> source src/constraints/RISC_CORE.tcl
```

After applying the top-level constraints, you have a constrained GTECH design database. The design is now ready for compilation.

Compiling the Design

Use the `acs_compile_design` command to compile the RISC_CORE design.

```
dc_shell-xg-t> acs_compile_design RISC_CORE
```

This command performs the following tasks:

1. Creates the pass0 directory to store the results (if it doesn’t already exist)
2. Resolves multiple instances

   In the RISC_CORE design, the STACK_MEM subdesign has multiple instances (I_STACK_TOP/I1_STACK_MEM, I_STACK_TOP/I2_STACK_MEM, and I_STACK_TOP/I3_STACK_MEM). Automated Chip Synthesis resolves the multiple instances of STACK_MEM by selecting I_STACK_TOP/I1_STACK_MEM as the master instance. For more information about master instances, see “Resolving Multiple Instances” on page 3-13.

3. Identifies the compile partitions

   Automated Chip Synthesis selects the first-level subdesigns, as well as any multiply instantiated designs, as compile partitions. The compile partitions for RISC_CORE are ALU, CONTROL, DATA_PATH, INSTRN_LAT, PRGRM_CNT_TOP, REG_FILE, STACK_TOP, and STACK_MEM. For more information about compile partitions, see “Specifying Compile Partitions” on page 3-18.

   Example 4-2 shows the partitions report (generated by the `report_partitions` command) for the unmapped RISC_CORE design. The partitions report also identifies multiple instances and their master instance.

Example 4-2 Partitions Report

```
 Loading design 'RISC_CORE'
*******************************************************************************
Report    : partitions
Design    : RISC_CORE (unmapped)
Date      : Wed Jul 6 14:59:22 2005
*******************************************************************************
Partition attributes :

  (*)   -  partition.
  %     -  the percentage w.r.t total design area.
  d     -  dont_touch.
  m(n)  -  multiple instantiated design (number of instantiation).

Warning: No area estimation numbers are available yet. (ACS-153)
Designs                                 Attributes
------------------------------------------------------------------------------
RISC_CORE(*)                            0.0%
    STACK_TOP(*)                        0.0%
        STACK_MEM(*)                    0.0%, m(3), I_STACK_TOP/I1_STACK_MEM
        STACK_FSM                       0.0%
    REG_FILE(*)                         0.0%
    PRGRM_CNT_TOP(*)                    0.0%
        PRGRM_CNT                       0.0%
        PRGRM_DECODE                    0.0%
        PRGRM_FSM                       0.0%
    INSTRN_LAT(*)                       0.0%
    DATA_PATH(*)                        0.0%
    CONTROL(*)                          0.0%
    ALU(*)                              0.0%
```

4. Generates partition constraints

   Automated Chip Synthesis uses RTL budgeting to generate constraints for each of the compile partitions. The partition constraint files are saved in the pass0/constraints directory. For more information about RTL budgeting, see “RTL Budgeting” on page 2-20.

5. Generates the partition compile scripts

   The partition compile scripts are saved in the pass0/scripts directory. For more information about the compile scripts, see “Generating the Compile Scripts” on page 2-24.

6. Generates the makefile to compile the design

   The makefile is saved in the pass0/Makefile file. For more information about the makefile, see “Generating the Makefile” on page 2-29.

7. Removes the existing designs from memory

8. Performs the parallel compile

   The compiled partitions are written to the pass0/db/post_compile directory in .ddc format.

9. Generates the HTML summary file ($acs_work_dir/ACS_results_RISC_CORE.html)

Analyzing the Compiled Design

The `acs_compile_design` command generates the following reports for the top-level design: area, timing, constraints, and quality of results. These reports are placed in the pass0/reports directory. You can access these reports through the HTML summary file. To see the timing report, view the $acs_work_dir/ACS_results.html file in your browser, click pass0, then click report_timing in the Reports section. Example 4-3 shows the timing report, which indicates that the design does not meet timing constraints. You must perform an incremental compile to fix the timing violations.

Example 4-3 Timing Report

```
****************************************
Report : timing
 -path full
 -delay max
 -max_paths 1
Design : RISC_CORE
Version: X-2005.09
Date   : Tue Jul 5 11:59:22 2005
****************************************
Operating Conditions:nom_pvt   Library: cba_core
Wire Load Model Mode: enclosed

  Startpoint: I_INSTRN_LAT/Crnt_Instrn_2_reg[30]
              (rising edge-triggered flip-flop clocked by Clk)
  Endpoint: RESULT_DATA[5]
            (output port clocked by Clk)
 Path Group: Clk
 Path Type: max
...

 -----------------------------------------------------------------------------
 data required time                                                   1.70
 data arrival time                                                   -2.38
 -----------------------------------------------------------------------------
 slack (VIOLATED)                                                    -0.68
```

Refining the Design

Use the `acs_refine_design` command to refine the RISC_CORE design.

```
dc_shell-xg-t> acs_refine_design RISC_CORE
```

This command performs the same tasks as the `acs_compile_design` command, with the following differences:

- It uses the pass1 directory for the generated files, instead of the pass0 directory.

- It performs design budgeting on the previously compiled design to generate the partition constraints, instead of using RTL budgeting on the GTECH design.

  For information about design budgeting, see “Generating Partition Constraints for Gate-Level Designs” on page 2-37.

- It performs an incremental compile, instead of a full compile.

  For information about the incremental compile scripts used by `acs_refine_design`, see “Generating the Compile Scripts” on page 2-40.

Analyzing the Refined Design

To see the pass1 timing report, view the $acs_work_dir/ACS_results_RISC_CORE.html file in your browser, click pass1, then click report_timing in the Reports section. The timing report shows that although the timing has improved, the design still does not meet timing constraints. You can use the following methods to try to fix these violations:

- Run the `acs_recompile_design` command.

  For information about this command, see “Recompiling Your Design” on page 3-52.

- Change the compile strategy.

  For information about changing the compile strategy, see “Changing the Compile Strategy” on page 3-29.

Index

A

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

U

V